VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Technologie Java Servlet 3.0

Java Servlet 3.0 Technology

2010

Bc. Přemysl Kaloč

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 6. května 2010 .....

Rád bych na tomto místě poděkoval všem, kteří mne při psaní této diplomové práce podporovali, zejména pak vedoucímu diplomové práce Ing. Štěpánu Kuchařovi za cenné rady a připomínky.

Abstrakt

Dne 10. prosince 2009 byla vydána prozatím nejnovější verze technologie Java Servlet, jež nese označení 3.0. Tato verze se od svých předchůdkyň liší, jelikož sebou nepřináší pouze jednu či dvě velké změny, ale hned několik zajímavých novinek. Mezi takovéto novinky například patří možnost asynchronního zpracování požadavků, fragmentace souboru web.xml, anotace pro definici servletů, filtrů a posluchačů či programově řízená autentizace uživatelů. Těmto novinkám, respektive nejnovější verzi servletů, je věnována první část této práce. Druhá část se pak zabývá návrhem a implementací vlastního servletového kontejneru, který využití několika z těchto novinek umožní.

Klíčová slova: servlet 3.0, servletový kontejner, Pike, historie servletů, servletová webová aplikace, JavaEE, java

Abstract

10 December 2009 was released the latest version of the Java Servlet technology, which is called 3.0. This version differs from its predecessors, since it does not bring just one or two big changes, but several innovations, such as the possibility of asynchronous request processing, modularization of the web.xml file, annotations for servlet, filter and listener definition, or programmatic user authentication. The first part of this thesis deals with these new features and with the newest version of the Java Servlet technology, respectively. The second part is devoted to design and implementation of own simple servlet container, which makes it possible to work with some of these innovations.

Keywords: servlet 3.0, servlet container, Pike, history of servlets, servlet web application, JavaEE, java

Seznam použitých zkratek a symbolů

AJAX	–	Asynchronous JavaScript and XML
API	–	Application programming interface
ASP	–	Active Server Pages
CD	–	Compact Disc
CDC	–	Connected Device Configuration
CGI	–	Common Gateway Interface
CSS	–	Cascading Style Sheets
EJB	–	Enterprise JavaBeans
HTML	–	Hyper Text Markup Language
HTTP	–	Hypertext Transfer Protocol
HTTPS	–	Hypertext Transfer Protocol Secure
IDE	–	Integrated development environment
IoC	–	Inversion of control
ISAPI	–	Internet Server Application Programming Interface
JCP	–	Java Community Process
JDBC	–	Java Database Connectivity
JSF	–	JavaServer Faces
JSP	–	JavaServer Pages
JSR	–	Java Specification Request
MIME	–	Multipurpose Internet Mail Extensions
MVC	–	Model–View–Controller
MIDP	–	Mobile Information Device Profile
NSAPI	–	Netscape Server Application Programming Interface
PDF	–	Portable Document Format

RFC	–	Request for Comments
RUP	–	Rational Unified Process
SSJS	–	Server-side JavaScript
SSL	–	Secure Sockets Layer
TCK	–	Technology Compatibility Kit
TCP	–	Transmission Control Protocol
UML	–	Unified Modeling Language
URL	–	Uniform Resource Locator
VBScript	–	Visual Basic Scripting Edition
XML	–	Extensible Markup Language

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Pokud vyvíjíte webové aplikace v Javě, zcela jistě jste už zaslechli slovo „*servlet*“. Java Servlet technologie je, aniž bychom si to my, Java vývojáři, možná uvědomovali, jedna z nejpoužívanějších technologií pro tvorbu webových aplikací vůbec. Nicméně asi většina z nás neprogramuje servlety přímo, ale využívá některý z frameworků, který nad servlety staví a programátorům umožňuje, co možná nejpohodlnější tvorbu webových aplikací. Mezi takovéto frameworky můžeme například zařadit Apache Struts, Spring Web MVC či Java Server Faces. Z těchto důvodů jsem přesvědčen, že porozumět servletové technologii může každého Java vývojáře jen obohatit a toto byla také jedna z hlavních příčin, proč jsem si pro mou diplomovou práci zvolil právě servlety.

Jako každá technologie se i servlety vyvíjí a dnes již existuje jejich celkově osmá verze s označením 3.0. Tato verze vyšla teprve nedávno (přesně 10. 12. 2009) a přinesla celou řadu zajímavých novinek, jimž je věnována celá třetí kapitola. Popis těchto novinek byl jedním z hlavních úkolů této práce. Dalším posláním byl návrh a následná realizace vlastního jednoduchého servletového kontejneru, který bude implementovat základní vlastnosti servletové technologie včetně těch, které nám přinesla verze 3.0. Tomuto kontejneru je věnována čtvrtá kapitola.

Aby tato práce byla kompaktní, rozhodl jsem se druhou kapitolu věnovat základům servletů. V této kapitole jsem se snažil vybrat to nejpodstatnější, aby i čtenáři neznalí servletů byli po přečtení této kapitoly schopni říci, co to servlety jsou a jak si pomocí nich naprogramovat jednoduchou webovou aplikaci. Inspirací pro tuto kapitolu mi byly dvě skvělé knížky [1, 2], kde především [2] doporučuji k přečtení.

2 Technologie Java Servlet

Hlavím cílem této kapitoly je, pro snazší pochopení následujících kapitol, vysvětlit princip fungování webových aplikací a popsat základy servletů. Je jasné, že obsah této kapitoly je určen pouze pro ty, kteří doposud servlety neznají a chtějí se je naučit. Pokud však servlety již ovládáte, můžete celou tuto kapitolu přeskočit a ihned začít třetí kapitolou, která pojednává o novinkách, které nám přinesla nedávno vydaná, nejnovější verze servletové specifikace.

Jistě chápete, že obsáhnout v jedné kapitole vše, co se týče servletů, není možné, a proto, pokud to se servlety myslíte opravdu vážně, doporučuji k přečtení skvělou knížku [2], která servlety vyučuje netradiční, ale velmi čtivou formou. Tato publikace byla mou hlavní inspirací při psaní této kapitoly.

2.1 Webové aplikace

Textovým dokumentům s HTML značkami se říká statické webové stránky. Toto označení je trefné, neboť kdykoliv o ně, webový server, požádáte, vždy vypadají a chovají se úplně stejně. Typickým příkladem takovéto stránky je například HTML specifikace na webu konsorcia W3C[1]. Kdykoliv na tuto adresu přejdete, vždy bude její obsah identický.

Statické stránky tady s námi jsou již od počátků internetu a jejich nedostatek je zřejmý. Co kdybychom například požadovali, aby nám stránka vypisovala aktuální datum a čas? Tento požadavek již pomocí statických stránek není možné splnit a tak musíme využít nějaké jiné řešení. Tímto řešením je generování webových stránek až na základě uživatelských požadavků, tedy vytvářet tzv. dynamické stránky. Například stránka, která vypisuje Váš emailový účet, nutně musí být dynamická, neboť v případě, kdy by Vám vypisovala pořád jedny a tentýž emaily byste s ní zřejmě moc spokojeni nebyli. Pro to, abychom byli schopni dynamické stránky vytvářet, nám již nestačí pouhý webový server, ale potřebujeme nějakou pomocnou aplikaci, která tento úkol bude řešit za něj. Takováto pomocná aplikace jsou například servlety, o kterých bude řeč později. V této chvíli nám stačí vědět pouze to, že kdykoliv klient požádá o dynamickou stránku, server daný požadavek neobslouží sám, ale ihned ho předá odpovídající pomocné aplikaci. Ta na jeho základě nějakým způsobem vygeneruje odpověď, následně ji předá webovému serveru nazpátek, a ten ji poté odešle klientovi. Lépe je tento proces patrný z obrázku 1.

Dříve, než přišly na řadu servlety, ustálilo se několik hojně využívaných technologií pro tvorbu dynamických stránek (tedy pomocných aplikací webových serverů). Těmito technologiemi se budou zabývat následující podkapitoly, abyste pak byli schopni ocenit, jaké výhody oproti těmto technologiím servlety mají.[2].

[1] HTML specifikace je dostupná pod URL http://www.w3.org/TR/1999/REC-html401-19991224/.

[2] Jsou popsány pouze technologie na straně serveru. Klientskými technologiemi, jako např. „JavaScriptem“, Adobe „Flashem“ či Java applety, se tato práce nezabývá. Také nejsou popsány ty technologie, které vznikly až po servletech, a jenž nejsou jejich pouhou nadstavbou. Především se jedná o technologii ASP.NET od společnosti Microsoft. ASP.NET se můžete naučit například zde: http://www.asp.net/.

Obrázek 1: Princip zpracování požadavku na dynamickou stránku

2.1.1 CGI

CGI byla jedna z prvních technologií pro tvorbu dynamických webových stránek. Paradoxně nebyla pro tento účel původně stvořena, ale postupem času se pro dynamické stránky začala využívat. CGI je vlastně protokol, který specifikuje, jakým způsobem bude server komunikovat s pomocnými aplikacemi. Pomocné aplikace zde jsou libovolné konzolové aplikace obvykle napsané v interpretovaném jazyce Perl, ale může být využito i jiných platforem. Vždy, když server příjme od klienta požadavek, spustí novou instanci odpovídající aplikace a následně jej pomocí vstupních parametrů (někdy také standardním vstupem či proměnným prostředím) vytvořené aplikaci předá. Ta následně vygeneruje odpověď, předá ji serveru pomocí standardního výstupu nazpátek a ten ji zase odešle klientovi.

Z výše uvedených informací vyplývá, že CGI byla technologií průlomovou, ale měla také celou řadu nedostatků týkajících se především výkonnosti (pro každý požadavek se vytváří nový proces) a způsobu předávání požadavků konkrétním pomocným aplikacím.

2.1.2 FastCGI

Nedostatky CGI byly natolik problematické, že netrvalo dlouho a vzniklo tzv. FastCGI. Tato technologie řešila především špatnou škálovatelnost výkonu CGI skriptů, jak se obvykle pomocným aplikacím při použití CGI říká. Zde se již u požadavků na stejný cíl nevytváří stále nové a nové procesy, ale je spuštěn pouze jeden proces, který všechny tyto požadavky následně obsluhuje. Také data požadavků se již nepředávají pomocí vstupních parametrů, ale se serverem je komunikováno přes standardní TCP rozhraní.

2.1.3 mod_perl

mod_perl není nová technologie jako taková (což ostatně není ani FastCGI), ale pouze jedno z dalších vylepšení CGI skriptů. mod_perl je vlastně programový modul webového serveru Apache[3], který zvyšuje rychlost CGI skriptů tím, že nabízí již inicializovaný interpret programovacího jazyka Perl, který se tak již při každém požadavku nemusí znovu nahrávat (samozřejmě, pokud je konkrétní CGI skript v „Perlu“ napsán). Krom toho také programátorům nabízí bohaté API webového serveru Apache.

[3] Apache HTTP Server je zřejmě nejznámější webový server na světě. Stáhnout si jej můžete z této URL adresy: http://httpd.apache.org/.

2.1.4 PHP

Troufám si tvrdit, že mnoho programátorů webových aplikací technologii mod_perl nikdy nepoužilo. Ovšem v případě PHP tomu bude zřejmě naopak, i když je to platforma technologii mod_perl velmi podobná. Vlastně se dá říci, že se liší pouze syntaxí skriptovacího jazyka (oproti „Perlu“ se používá vlastní skriptovací jazyk) a programovým API. Vše ostatní, jako interpretace skriptů a jeden interpret pro všechny pomocné aplikace, zůstalo zachováno.

Právě ono programové API stálo za úspěchem PHP. Toto API totiž, již velmi brzo po svém uvedení, obsahovalo mnoho funkcionalit, které programátorů ulehčovaly práci. Například automatickou správu formulářových proměnných, syntaxi pro pohodlné vkládání HTML, podporu komunikace s databází a mnoho dalších.

Je ale nutné říci, že PHP technologie měla také své nevýhody. Ty se týkaly především chybovosti API a bezpečnosti PHP skriptů.

2.1.5 NSAPI a ISAPI

Jelikož je CGI standard, může být implementováno libovolným tvůrcem webového serveru. I díky tomuto faktu se již brzy po svém uvedení stalo velmi oblíbené. Nicméně i přesto postupem času přišli jednotliví výrobci webových serverů se svými specializovanými řešeními, které ale všechny měly jednu nevýhodu, fungovaly pouze na konkrétních webových serverech. Nejznámější z těchto řešení byly NSAPI a ISAPI, což jsou programové rozhraní od společnosti Netscape, respektive Microsoft, které dovolují programátorům naimplementovat vlastní techniku pro obsloužení požadavků na dynamické stránky. Vytvořená funkcionalita je však velice úzce spjatá s daným serverem, ale zase může být plně využito jeho nabízených možností.

2.1.6 SSJS a ASP

Jak Netscape, tak Microsoft, krom výše uvedených serverových API, nabízeli programátorům další techniku pro tvorbu dynamických stránek. Je to možnost vkládat malinké kusy kódu přímo do statického HTML, které je pak čteno a vykonáváno speciální logikou. Ta každý kód nejprve spustí, ten vygeneruje textový výstup, jímž je pak daný kód nahrazen. Tohoto principu se využívá dodnes.

U SSJS se do textu s HTML značkami vkládá Javascript, u ASP to je většinou VBScript.

2.2 Základy Java Servlet technologie

Již z názvu této podkapitoly vyplývá, že popisuje základy servletů. K tomu, abyste se tyto základy lépe naučili, je vhodné si ze stránek JCP[4] stáhnout nejnovější verzi servletové

[4]Java Community Process (založený roku 1998) je formalizovaný proces, který všem zájemcům dovoluje podílet se na vývoji budoucích specifikací celé Java platformy. Specifikacemi například jsou již zmíněné servlety, dále pak API pro zpracování XML, EJB, CDC, MIDP, JSP, JDBC a mnoho dalších. Klíčovým prvkem jsou tzv. JSR dokumenty, které jednotlivé specifikace obsahují. Tyto dokumenty se vytvářejí pro každou verzi dané technologie a tvoří je expertní skupina, což je volená množina určitých firem, popřípadě jednotlivců.

specifikace. Toto učiníte tak, že přejdete na stránku pro JSR 315[5], kliknete na hypertextový odkaz s textem „*Download*“, budete souhlasit s licencí a následně stáhnete všechny tři nabízené soubory. První soubor je samotný text specifikace ve formě PDF dokumentu, druhý a třetí jsou ZIP archívy se „Servletovým API“ a jeho dokumentací ve formě Javadoc, kterou při čtení následujících odstavců doporučuji mít otevřenou.

2.2.1 Přednosti servletů

Než pokročíme k základům servletů, je vhodné nastínit, jaké výhody má tato technologie oproti výše zmíněným technikám:

- Bezpečnost. Jelikož nejsou servlety nic jiného než pouhé Java třídy, jsou spravovány virtuálním strojem, který jim nedovolí provádět nic víc, než jejich jakékoliv běžné kolegyni. Například u technologií NSAPI/ISAPI byla bezpečnost na poměrně nízké úrovni, jelikož veškerá programátory doimplementovaná logika běžela ve stejném procesu jako server a taktéž byla kompilována vůči jeho dynamickým knihovnám. Z tohoto důvodu mohla jakákoliv chyba „shodit“ celý server.

- Přenositelnost. Tato výhoda plyne již z pouhé povahy platformy Java a existence servletů jako standardu. Nejenom, že Vaše webová aplikace bude přenositelná na různé platformy podporující Javu, ale také ji budete moci nasadit na libovolný webový server implementující servlety, tzv. „*servletový kontejner*“. Toto u různých proprietárních řešení možné není.

- Snadnost implementace. Jelikož jsou servlety Java třídy, lze je velmi snadno a elegantně implementovat a navíc můžete využít bohaté API platformy Java.

- Výkon. Servlety mohou být velmi rychlé, neboť nejsou interpretované, po inicializaci obvykle zůstávají stále v paměti a každý servlet je využit pro obsluhování všech následných požadavků na konkrétní webový zdroj. Nejsou zde vytvářeny žádné další procesy (dnes už ani vlákna) a pokud je servlet správně naimplementován, může za krátký časový okamžik obsloužit mnoho uživatelských požadavků.

2.2.2 Nedostatky servletů

Jelikož jsem uvedl přednosti servletů, je vhodné zmínit také jejich nedostatky:

- Paměťová náročnost. Poněvadž jsou servlety drženy v paměti, může, při jejich opravdu velkém počtu nebo paměťově náročných instančních proměnných, dojít webovému serveru operační paměť. Toto je problém například u klasických Java hostingů, neboť ty obvykle mají menší počet serverů a velký počet nahraných webových aplikací. Oproti tomu, u lepších poskytovatelů s virtuálními servery a

[5]Specifikace JSR 315 je dostupná na této URL adrese: http://jcp.org/aboutJava/communityprocess/final/jsr315/index.html.

virtualizovanými zdroji se tento problém stírá. Také samotný servletový kontejner a jeho webové aplikace mohou být distribuovány přes vícero počítačů.[6]

- Nepohodlné vytváření HTML výstupu (obecně jakýchkoliv formátovaných dat). Samotné servlety nenabízejí žádnou možnost, která by ulehčila generování statického obsahu. Tato funkcionalita se musí řešit až ve vyšších vrstvách, nebo speciálními aplikacemi, které automaticky vytvářejí kód servletů.[7]
- Špatná separace byznys logiky od uživatelského rozhraní. Servlet technologie nebyla navrhována s myšlenkou oddělení logiky aplikace od jejího rozhraní na mysli. Pokud tento nedostatek programátor neřeší sám, nebo nevyužívá služeb některé z knihoven, servlet obvykle obsahuje mix HTML a vlastní logiky.[8]

2.2.3 Co je to servlet?

Servlet je libovolná třída programovacího jazyka Java, která implementuje rozhraní *Servlet* z balíčku *javax.servlet*. Programovat servlety je tedy stejně snadné, jako implementovat jakékoliv jiné třídy, které rozšiřují a využívají nějaké knihovní API.

Každý servlet nahrazuje jednu výše zmíněnou pomocnou aplikaci, tedy slouží k obsluhování všech požadavků na konkrétní URL. Obsluha klientských požadavků je realizována metodou *service(ServletRequest, ServletResponse)* z rozhraní *Servlet*, která je pro každý požadavek automaticky volána. Tedy k tomu, abyste přijaté požadavky zpracovali, musíte tuto metodu nejprve implementovat. Obsluha většinou probíhá tak, že jsou, přes obdržený objekt typu *ServletRequest*, nejprve načtena data požadavku a následně je na základě těchto dat naplněn objekt typu *ServletResponse*, který zde reprezentuje odpověď klientovi. Poté, co je tato metoda vykonána, je klientovi naplněný objekt automaticky odeslán nazpátek.

Jednotlivé servlety tedy programujeme vůči tzv. „*Servlet API*“, které je z velké části reprezentováno balíčky *javax.servlet* a *javax.servlet.http*. Balíček *javax.servlet* definuje všechny základní rozhraní a třídy, které dohromady tvoří kostru servletového API, a jenž musíme, za účelem obsluhy požadavků, nejprve rozšířit. Naštěstí toto pro servlety, které budeme využívat pro zpracování požadavků protokolu HTTP, provádět již nemusíme, jelikož servletové API toto rozšíření prostřednictvím balíčku *javax.servlet.http* již nabízí. Implementace HTTP servletů je tedy pohodlnější, neboť již například není nutné přímo implementovat rozhraní *Servlet*, ale stačí, když pro realizaci servletů budeme rozšiřovat třídu *javax.servlet.HttpServlet*, která pro obsluhu jednotlivých typů HTTP požadavků definuje

[6] Například v technologii ASP.NET nejsou jednotlivé „Web Forms“, které, na této platformě, můžeme chápat jako takové servlety, vůbec v paměti drženy, nýbrž se při každém požadavku znovu vytvářejí, respektive jejich instance.

[7] Tento problém například řeší tzv. JSP stránky, které do Java světa přinášejí koncept SSJS, respektive ASP technologií. Pokud JSP technologii doposud neznáte, můžete se podívat přímo do specifikace, která je dostupná v rámci JSR 245 pod touto URL: http://jcp.org/en/jsr/detail?id=245.

[8] Například technologie ASP.NET již od svých počátků tento problém řešila a výsledkem je tzv. „Code-Behind“ koncept, o kterém se více můžete dozvědět zde: http://en.wikipedia.org/wiki/ASP.NET#Code-behind_model.

samostatné metody. Tedy, pokud například chceme, aby byl náš servlet schopen reagovat na GET a POST požadavky, jednoduše překryjeme metody *doGet(HttpServletRequest, HttpServletResponse)* a *doPost(HttpServletRequest, HttpServletResponse)*. Jistě jste si také všimli, že tyto metody již nepřijímají pouhé *ServletRequest* a *ServletResponse* objekty, ale objekty tříd *HttpServletRequest* a *HttpServletResponse*, což jsou jejich HTTP kolegyně, které je rozšiřují o mnoho, HTTP protokolu, specifických funkcionalit. Kdykoliv je tedy přijat GET či POST požadavek, je zavolána metoda *doGet*, respektive *doPost*.

Ve výpisu 1 můžete vidět implementaci jednoduchého HTTP servletu, který na všechny GET a POST požadavky odpoví HTML stránkou s textem „*Hello World!*“. Tento servlet tedy rozšiřuje třídu *HttpServlet* a překrývá metody *doGet* a *doPost*, jejichž implementace jsou naprosto identické, jelikož zde nepotřebujeme mezi GET a POST požadavky nijak rozlišovat. Prvním příkazem je pomocí metody *setContentType(String)* nastaven MIME typ odpovědi (zde je MIME typ nastaven na *text/html*, jelikož odesíláme HTML stránku) a následně je pomocí objektu *HttpServletResponse* získán *PrintWriter*, pomocí něhož je pak zapsán požadovaný HTML dokument.

```
public class HelloWorldServlet extends HttpServlet {

    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println ("<html><head>");
            out.println ("<title>Hello World!</title></head>");
            out.println ("</head><body>");
            out.println ("<h1>Hello World!</h1>");
            out.println ("</body></html>");
        } finally {
            out.close();
        }
    }

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      processRequest(request, response);
    }

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
      processRequest(request, response);
    }

}
```

Výpis 1: Jednoduchý HTTP servlet

2.2.4 Servletový kontejner

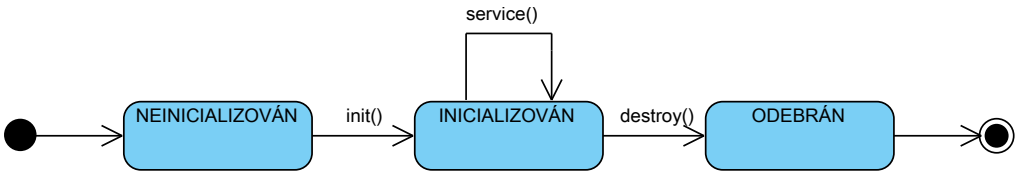
Z rozhraní *javax.servlet.Servlet* je zřejmé, že servlety oproti klasickým Java aplikacím nemají žádnou metodu *main(String[])*. Oni vlastně žádnou takovouto metodu nepotřebují, jelikož nejsou spouštěny uživatelem, ale něčím, co se nazývá „*servletový kontejner*".

Specifikace říká, že servletový kontejner je část webového serveru, který implementuje *Servlet API* a ctí servletovou specifikaci. Jeho hlavní funkce jsou tedy následující:

- Podpora komunikace. K realizaci webových aplikací nám stačí znát pouhé *Servlet API* a následně s jeho pomocí realizovat požadovanou byznys logiku. Vše ostatní zařídí webový server a servletový kontejner, na kterém budou naše aplikace nahrány.
- Správa životního cyklu servletů. Rozhraní *Servlet* specifikuje celkem tři metody, které reprezentují životní cyklus servletů. O volání těchto metod, a tedy o správu životního cyklu, se stará servletový kontejner. Životní cyklus servletů je podrobněji popsán v následující podkapitole.
- Simultánní obsluha požadavků. Pro každý servlet je vytvořena pouze jedna instance, která je pak využita pro obsluhu všech požadavků, které jsou pro ni určeny. Aby tedy bylo možné obsloužit více než jeden požadavek najednou, je pro každý nově příchozí požadavek vytvořeno nové vlákno, kterým je pak následně volána metoda service nad danou instancí servletu. Může se tedy stát, že je metoda service v libovolnou dobu vykonávána mnoha vlákny, jelikož byl daný servlet v danou chvíli požadován mnoha klienty najednou.

Při využití servletového kontejneru obsluha, jednoho HTTP klientského požadavku na konkrétní servlet, probíhá zhruba tímto způsobem:

1. Uživatel (obvykle pomocí webového prohlížeče) zašle webovému serveru HTTP požadavek.
2. Webový server tento požadavek předá servletovému kontejneru.
3. Servletový kontejner zjistí, že je požadavek určen servletu, a tak vytvoří dva objekty: *HttpServletRequest* a *HttpServletResponse*. První objekt je objektová reprezentace přijatého HTTP požadavku, druhý slouží ke generování HTTP odpovědi.
4. Servletový kontejner získá referenci na daný servlet a v případě, že doposud žádná jeho instance neexistuje, bude vytvořena.
5. Servletový kontejner spustí nové vlákno, kterému následně předá vytvořené dva objekty a uchovanou referenci na požadovaný servlet.
6. Vytvořené vlákno volá metodu *service* (ta, dle typu požadavku, zase volá některou z *doXXX* metod. Například *doGet*.) nad přijatým servletem. Metodě *service* jsou jako parametry předány obdržené *HttpServletRequest* a *HttpServletResponse* objekty.



Obrázek 2: Životní cyklus servletu

7. Ve voláné *doXXX* metodě je vygenerována odpověď, která je následně zapsána do přijatého *HttpServletResponse* objektu.
8. Metoda *doXXX* končí a tak končí i vytvořené vlákno.
9. Servletový kontejner transformuje naplněný *HttpServletResponse* objekt do surové HTTP odpovědi, kterou následně předá webovému serveru.
10. Webový server zašle vygenerovanou HTTP odpověď zpět klientovi.

2.2.5 Životní cyklus servletů

Každý servlet během své existence projde několika stavy, které definují jeho životní cyklus. Do jednotlivých stavů se dostane tak, že je na něm servletovým kontejnerem zavolána jedna ze tří specifických metod, které jsou definovány v rozhraní *javax.servlet.Servlet*. Jsou to metody *init(ServletConfig)*, *service(ServletRequest, ServletResponse)* a *destroy()*.

Na obrázku 2 je zobrazen stavový diagram, na kterém můžete vidět všechny stavy, jimiž každý servlet postupem času projde. Z diagramu vyplývá, že prvním stavem každého servletu je stav *neinicializován*. V tomto stavu se servlety nacházejí proto, že doposud na jejich instancích nebyla zavolána metoda *init*. K tomu, aby mohla být tato metoda vykonána, musí servletový kontejner nejprve nahrát bytecode servletových tříd do paměti a následně vytvořit jejich instance. Kdy se tento proces odehrává, záleží na dané implementaci servletovém kontejneru, neboť specifikace toto nijak nenařizuje. Typicky jsou instance vytvářeny buď všechny na začátku (tj. při startu servletového kontejneru), nebo jednotlivě až při prvních klientských požadavcích. Metoda *init* je volána pouze jednou a slouží k provedení všech jednorázových aktivit, které je pro daný servlet nutné provést, jako například vytvořit spojení k databázi apod. Dalším důležitým faktem je, že dokud není nad servlety tato metoda zavolána (nebo proběhne s chybou), nejsou servlety brány jako použitelné, a tudíž na ně není směřován žádný klientský požadavek. Jakmile je však *init* vykonána, servlet přejde do stavu *inicializován* a v tu chvíli může začít obsluhovat požadavky klientů, respektive servletový kontejner na něm může volat metodu *service*. Poslední stav, ve kterém se servlet může nacházet, je stav *odebrán*. Do tohoto stavu servlet přejde, jakmile je na něm vykonána metoda *destroy*. Tato metoda, stejně jako metoda *init*, je volána pouze jednou a její vykonání značí, že byl servlet deaktivován a již na něho nebudou směřovány žádné další požadavky. Obvykle je tato metoda volána v okamžiku ukončování servletového kontejneru.

2.2.6 Struktura webové aplikace

Servlety se na servletový kontejner nenahrávají samostatně, ale v rámci celých webových aplikací. Webové aplikace nejsou nic jiného, než kolekce různých typů souborů od servletů, přes obrázky až po statické HTML stránky.

Jelikož jsou však webové aplikace kolekcemi vícero souborů, musíme mít možnost říci, co je a co není servlet a být schopni jednotlivým souborům přiřadit konkrétní URL. Z těchto důvodu, servletová specifikace stanovuje určitá pravidla, která musíme při vytváření servletových webových aplikací dodržet:

- Všechny soubory webové aplikace musí být obsaženy v jedné jediné složce (tzv. *hlavní složce*).
- Hlavní složka musí obsahovat složku *WEB-INF*.
- Ve *WEB-INF* se musí nacházet XML soubor s názvem *web.xml* (tzv. „*deployment descriptor*“). Tento soubor slouží k přiřazení URL pro jednotlivé servlety a je mu věnována následující podkapitola.
- Všechen Java kód (včetně servletů) musí být obsažen ve složce *classes*, kterou musíme vytvořit ve *WEB-INF*.
- Pokud využíváme JAR soubory, musíme je uložit do složky *lib*, která se taktéž musí nacházet ve složce *WEB-INF*.
- Všechny ostatní soubory mohou být umístěny kdekoliv v *hlavní složce*.

K výše zmíněným pravidlům dále platí:

- Všem souborům, které umístíme mimo složku *WEB-INF*, bude přiděleno konkrétní URL, pod kterým bude zveřejněn jejich obsah. Typicky se zde ukládají veškeré statické HTML stránky, kaskádové styly, obrázky, videa a vůbec vše, co chceme, aby bylo uživatelům veřejně dostupné. V žádném případě zde neumisťujeme jakékoliv citlivé informace typu hesla k účtům, bankovní spojení apod. URL k souborům jsou vytvořeny tak, že se zřetězí URL *hlavní složky* (to je pevně dáno typem a aktuální konfigurací servletového kontejneru) a relativní cesta k daným souborům.
- Složka *WEB-INF* a veškeré soubory v ní obsažené jsou chráněny servletovým kontejnerem, respektive nevede k nim žádné URL. Jelikož jsou v této složce umístěny také servlety, složka navíc obsahuje soubor *web.xml*, pomocí nějž lze jednotlivým servletům přiřadit námi zvolené URL.
- Veškeré Java třídy a JAR soubory obsažené ve *WEB-INF* složce jsou nahrávány jedním „*ClassLoaderem*“. V servletech tedy můžeme využívat libovolný kód, který jsme do této složky umístili.

Všechny tyto pravidla musíme dodržet z jednoho prostého důvodu, každý servletový kontejner výše zmíněnou strukturu webových aplikací předpokládá a ví, jak s ní zacházet. Pokud bychom ji nedodrželi, zřejmě by se naše aplikace nechovala očekávaným způsobem anebo by nefungovala vůbec.

2.2.7 Deployment descriptor

Již víme, že *deployment descriptor* je XML soubor, pomocí nějž přiřazujeme URL k jednotlivým servletům. Možnosti tohoto souboru jsou ovšem daleko větší, neboť je to centrální konfigurační prvek každé servletové webové aplikace. Navíc z jeho XML povahy vyplývá, že konfigurace probíhá jednoduchým deklarativním způsobem.

Pomocí *deployment descriptoru* můžeme (krom jiných) nastavit tyto položky:

- Deklarovat, které třídy ve složce *WEB-INF* má servletový kontejner chápat jako servlety (tyto třídy musí samozřejmě implementovat rozhraní *javax.servlet.Servlet*).
- Přiřazovat URL jednotlivým servletům.
- Stanovit, zda se mají konkrétní servlety nahrávat již při startu servletového kontejneru, nebo až při prvním požadavku, jako je to obvyklé.
- Specifikovat vstupní parametry jednotlivých servletů (hodnoty těchto parametrů pak mohou být načteny pomocí *Servlet API*).
- Definovat tzv. *„filtry“*, o kterých se letmo zmiňuji v následující podkapitole, jenž je věnována popisu hlavních částí servletového API.
- Specifikovat posluchače pro různé typy událostí. Například pro zjištění, že je naše aplikace aktuálně startována servletovým kontejnerem. Některé typy posluchačů taktéž popisuji v následující podkapitole.
- Stanovit chybové stránky pro situace, kdy vykonávaní jednotlivých požadavků skončí s chybou. Namísto chyby je uživatelům vrácen obsah těchto stránek.
- Deklarovat různá bezpečnostní omezení. Například kdo jaký servlet může využít, zda musí být komunikace šifrována atp.
- Nastavit parametry pro HTTP relace.

Syntaxe deployment descriptoru je definována pomocí XML schématu, které je popsáno jak v servletové specifikaci, tak dostupné samostatně prostřednictvím JSR 315 na stránkách JCP. Na výpisu 2 můžete vidět ukázkový *deployment descriptor* (pro webovou aplikaci s názvem *„webapp1“*), který specifikuje dva servlety, předává jim inicializační parametry a mapuje je na dané URL.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.org
    /2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://
    java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <display-name>webapp1</display-name>
  <servlet>
    <servlet-name>HelloWorldServlet1</servlet-name>
    <servlet-class>HelloWorldServlet1</servlet-class>
    <init-param>
```

```
            <param-name>WorldName</param-name>
            <param-value>Earth</param-value>
        </init-param>
    </servlet>
    <servlet>
        <servlet-name>HelloWorldServlet2</servlet-name>
        <servlet-class>HelloWorldServlet2</servlet-class>
        <init-param>
            <param-name>GreetingType</param-name>
            <param-value>Hello</param-value>
        </init-param>
        <init-param>
            <param-name>WorldName</param-name>
            <param-value>Mars</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloWorldServlet1</servlet-name>
        <url-pattern>/HelloWorldServlet1</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>HelloWorldServlet2</servlet-name>
        <url-pattern>/HelloWorldServlet2</url-pattern>
    </servlet-mapping>
</web-app>
```

Výpis 2: Ukázka *deployment descriptoru*

2.2.8 Servlet API

Servlet API je běžné Java API, které definuje všechny potřebné rozhraní a třídy nutné k implementaci servletových webových aplikací. Toto API implementuje každý servletový kontejner, tedy při jeho využití nemůže docházet k chybám typu *ClassNotFoundException* a podobným. Pro to, abyste jej ve Vašich třídách mohli použít, musíte si ze stránek JCP v rámci JSR 315 nejprve stáhnout JAR soubor s jeho referenční implementací a Javadoc. Implementace jsou také dostupné v rámci jednotlivých distribucí servletových kontejnerů.

Jak jsem již zmiňoval, kostru tohoto API tvoří dva Java balíčky, a to *javax.servlet* a *javax.servlet.http*. Účel jejich hlavních tříd popisuji v následujících odstavcích.

2.2.8.1 Typy Servlet, GenericServlet a HttpServlet

Aby se libovolná Java třída proměnila v servlet, a bylo s ní servletovým kontejnerem takto zacházeno, musí nejprve implementovat rozhraní *Servlet*. Toto rozhraní lze realizovat buď přímo, nebo nepřímo rozšířením tříd, které rozhraní *Servlet* již implementují. *Servlet API* takovéto třídy nabízí dvě, a to *javax.servlet.GenericServlet*, která je určena pro vytváření obecných servletů, a *javax.servlet.http.HttpServlet* pro servlety komunikující prostřednictvím HTTP.

Rozhraní *Servlet* definuje základní funkcionalitu servletů, tedy inicializaci, obsluhu klientských požadavků a destrukci. Abstraktní třída *GenericServlet* toto rozhraní imple-

mentuje a ponechává pouze jedinou abstraktní metodu, a to metodu *service*. Navíc také realizuje rozhraní *javax.servlet.ServletConfig*, o kterém bude řeč později.

Pokud jsou ale implementovány HTTP servlety, je nejlepší volbou abstraktní třída *HttpServlet*, která *GenericServlet* rozšiřuje o již zmíněné *doXXX* metody. Její hlavní funkcí je tedy implementace metody *service*, která pouze otestuje typ přijatého HTTP požadavku a na jeho základě následně zavolá jednu z metod *doXXX*, například *doGet*. Při použití *HttpServlet* tedy stačí, aby se překryla jedna z těchto metod a následně se v ní realizovala požadovaná byznys logika.

2.2.8.2 Rozhraní ServletConfig a ServletContext Pokud se podíváte do dokumentace, můžete vidět, že inicializační metoda servletů přijímá objekt typu *ServletConfig*. Přes tento objekt se servlety mohou dostat ke svým inicializačním parametrům, které lze deklarativním způsobem specifikovat v deployment descriptoru, a k objektu typu *ServletContext*.

Pokud rozhraní *Servlet* implementujete přímo, musíte si objekt *ServletConfig* uchovat, protože jste jej nuceni vracet metodou *getServletConfig()*, která je v rozhraní *Servlet* taktéž definována. Avšak pokud pouze rozšiřujete *GenericServlet* (popřípadě *HttpServlet*), toto uchování již provádět nemusíte, jelikož *GenericServlet* tento úkol realizuje za Vás a navíc, pro elegantnější přístup k inicializačním parametrům, rozhraní *ServletConfig* přímo implementuje, kde v jednotlivých metodách jen volá odpovídající metody na obdrženém *ServletConfig* objektu.

Objekt typu *ServletContext* je velmi důležitým společníkem každého servletu, poněvadž jsou servlety přes něj schopny komunikovat se servletovým kontejnerem a získávat tak různé kontextové informace. Kontextové z toho důvodu, že je pro každou webovou aplikaci vytvořena pouze jedna instance tohoto typu, která pak slouží jako jakási její reprezentace.

Přes rozhraní *ServletContext* lze získat/nastavit například tyto informace:

- Inicializační parametry definované pro celou webovou aplikaci.
- Část URL, která byla servletovým kontejnerem pro webovou aplikaci pevně zvolena (typicky je to název hlavní složky).
- Atributy, které jsou dostupné všem servletům webové aplikace (atributem může být libovolný objekt).
- Objekt typu *javax.servlet.RequestDispatcher*, pomocí kterého můžeme generování odpovědi přenechat jinému servletu, popřípadě vložit výstup jiného servletu do aktuálně generované odpovědi.
- Obsah libovolného souboru webové aplikace.
- Absolutní cestu k danému zdroji webové aplikace.
- Logovat libovolnou zprávu.

2.2.8.3 Rozhraní ServletRequest, HttpServletRequest, ServletResponse a HttpServletResponse

Již z názvů těchto rozhraní vyplývá, k jakému účelu slouží. Objekty implementující rozhraní *javax.servlet.ServletRequest*, respektive *javax.servlet.http.HttpServletRequest* jsou servlety přijímány v metodě *service* jako reprezentanti klientských požadavků. Servlet z nich může načíst například MIME typ požadavku, jeho velikost, kódování, obsah, parametry, použitý protokol a v neposlední řadě také IP adresu a port klienta. Přes rozhraní *HttpServletRequest* lze navíc získat cookie, názvy a obsah jednotlivých HTTP hlaviček, použitou HTTP metodu, specifické části URL a objekt typu *javax.servlet.http.HttpSession*, který reprezentuje HTTP relace.

Oproti tomu objekty typu *javax.servlet.ServletResponse* či *javax.servlet.http.HttpServletResponse* jsou servlety metodou *service* přijímány z toho důvodu, aby pomocí nich mohl servlet zapsat svou odpověď klientovi. Hlavní funkcionalitou těchto objektů je nastavit jednak typ odpovědi a její kódování, tak získat *OutputStream* (popřípadě *PrintWriter*), pomocí něhož je požadovaná odpověď zapisována. Možnosti rozhraní *HttpServletResponse* jsou o něco větší, kde dále můžeme nastavit například cookie data, HTTP hlavičky a číselný kód HTTP odpovědi. Také je možné využít pomocných metod pro zaslání chybových zpráv, popřípadě pro přesměrování klienta na jinou URL.

2.2.8.4 Třídy ServletInputStream a ServletOutputStream

Abstraktní třída *javax.servlet.ServletInputStream* slouží k načtení dat z klientského požadavku a lze ji získat zavoláním metody *getInputStream()* na objektech typu *ServletRequest*. Tato třída rozšiřuje *java.io.InputStream* o metodu *readLine(byte[], int, int)*, která načte jeden řádek ze vstupu, tedy všechny dostupné data po znak nového řádku.

Objekty typu *javax.servlet.ServletOutputStream* vrací metoda *getOutputStream()* v rozhraní *ServletResponse*. Tyto objekty využíváme pro zasílání libovolných dat konkrétním klientům. *ServletOutputStream* dědí od *java.io.OutputStream* a rozšiřuje jej o množinu metod *print()* a *println()*, pomocí nichž je možné zaslat jak všechny primitivní datové typy, tak také řetězce.

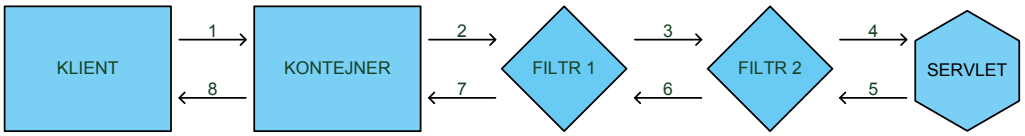
2.2.8.5 Rozhraní HttpSession

javax.servlet.http.HttpSession objekty můžete získat voláním metody *getSession()* na typech *HttpServletRequest*. Servletová specifikace těmito objekty realizuje paradigma HTTP relací, tzn., že pro každého klienta Vaší webové aplikace je vytvořen právě jeden takovýto objekt, který jej identifikuje. Alfou a omegou těchto objektů je, že si do nich servlety mohou ukládat libovolná data a tak například realizovat přihlašování, nákupní košík atp. Do relací je vhodné ukládat serializovatelné objekty, nebot' by v případě velkých dat a mnoha klientů nemusela serveru stačit paměť.

HTTP relace jsou specifikovány v dokumentu RFC 2109 a nověji také v RFC 2965.

2.2.8.6 Rozhraní Filter, FilterChain a FilterConfig

Rozhraní *javax.servlet.Filter* definuje tzv. *„filtry"*, které, stejně jako servlety, můžeme využít ke zpracování klientských požadavků a generování odpovědí. Hlavním rozdílem je, že toto zpracování je prováděno ještě předtím, než se dané požadavky k servletům vůbec dostanou, nebo poté, co



Obrázek 3: Princip zpracovávání požadavků s využitím filtrů

je servlety vygenerována odpověď, která ale doposud nebyla odeslána odpovídajícím klientům.

Filtry jsou tedy využívány k zachycení klientských požadavků a servletových odpovědí. Jejich definici provádíme v *deployment descriptoru*, kdy ke každému servletu můžeme definovat jejich libovolný počet. Hlavní výhodou filtrů je, že, aniž bychom museli dané servlety přeprogramovat, můžeme k nim přidat novou funkcionalitu.

Například testování, zda jsou uživatelé přihlášeni, bychom pomocí filtrů mohli realizovat takto:

1. Vytvoříme implementaci rozhraní *Filter*, která si z přijatého požadavku získá objekt *HttpSession* a v něm otestuje danou logickou proměnnou, která vyjadřuje stav přihlášení daného uživatele. Pokud je hodnota pravda, filtr následně zavolá metodu *doFilter(ServletRequest, ServletResponse)* na obdrženém objektu typu *javax.servlet.FilterChain*, která způsobí to, že je daný požadavek dále předán odpovídajícímu servletu (instance rozhraní *FilterChain* je servletovým kontejnerem vytvářena pro každý seznam filtrů daného servletu a je každému filtru předána v metodě *doFilter*, která je obdobou metody *service* u servletů. Filtry jsou pomocí tohoto objektu schopni řídit, zda se požadavek či odpověď předá dalšímu prvku v seznamu.). Pokud však uživatel přihlášen není, požadavek se servletu nepředá, nýbrž je uživateli vrácena webová stránka s přihlašovacím formulářem, kterou obsluhuje servlet, jenž v aplikaci realizuje přihlašování, a tedy nastavuje danou logickou proměnnou.

2. V *deployment descriptoru* filtr deklarujeme pro každý servlet, jehož použití je dovoleno pouze přihlášeným uživatelům.

Na obrázku 3 můžete vidět cestu požadavku a odpovědi, jenž patří servletu, pro který byly definovány dva filtry.

2.2.8.7 Rozhraní ServletContextListener, ServletContextAttributeListener

Pomocí těchto rozhraní je možné reagovat na události týkající se konkrétního *ServletContext* objektu. Lze reagovat na inicializaci či ukončování dané webové aplikace, respektive na změny atributů.

Implementace těchto rozhraní musí programátor deklarovat v *deployment descriptoru* pomocí XML elementu *listener*, ve kterém je nutno zadat plně kvalifikovaný název implementační třídy.

2.2.8.8 Rozhraní ServletRequestListener, ServletRequestAttributeListener

Implementacemi těchto rozhraní je programátor schopen reagovat na všechny příchozí a odchozí požadavky pro danou webovou aplikaci a na různé změny v atributech daného požadavku.

Implementace těchto rozhraní musí být taktéž deklarovány v *deployment descriptoru*.

2.2.8.9 Rozhraní HttpSessionListener, HttpSessionAttributeListener, HttpSessionBindingListener, HttpSessionActivationListener

Pomocí rozhraní *javax.servlet.http.HttpSessionListener*, respektive *javax.servlet.http.HttpSessionAttributeListener* lze reagovat na události týkající se vytváření a rušení HTTP relací, respektive změn atributů dané relace. Realizace těchto rozhraní musíme uvést v *deployment descriptoru*.

Rozhraní *javax.servlet.http.HttpSessionBindingListener* a *javax.servlet.http.HttpSessionActivationListener* musí implementovat jednotlivé atributy, které chtějí být notifikovány, že relace, ve které jsou uloženy, bude aktivována nebo deaktivována nebo že byly z dané relace programátory odebrány.

2.2.9 Nahrávání webové aplikace na servletový kontejner

Servletová specifikace nijak nenařizuje, jakým způsobem musí být webové aplikace na konkrétní servletové kontejnery nahrávány. Pouze stanovuje to, jakou strukturu (viz podkapitola 2.2.6) musí každá webová aplikace mít a nařizuje, že každý servletový kontejner je nucen ji přijímat. Doposud jsem však nezmínil, že webové aplikace můžeme také zabalit do ZIP archívu s koncovkou war, kde i tato forma musí být servletovými kontejnery akceptována.

Většina servletových kontejnerů proces nahrávání realizuje prostým překopírováním dané složky s webovou aplikací do specifické složky ve své distribuci. Obvykle je tato složka pojmenována „*webapps*“, což česky znamená „webové aplikace“. Tedy k tomu, abychom svou aplikaci na servletový kontejner nahráli, nám jednoduše stačí zkopírovat složku s naší aplikací do složky „*webapps*“ v kontejneru.

2.2.10 Ukázková servletová webová aplikace

Pro ukázku servletů v praxi jsem naprogramoval jednoduchou servletovou webovou aplikaci, která realizuje přihlašování uživatelů a jejich domovskou stránku. Podrobný popis této aplikace je uveden v příloze A.

3 Novinky ve specifikaci Java Servlet 3.0

V příloze B je sepsána krátká historie servletů a umístěn soupis největších změn v jednotlivých verzích servletové specifikace do verze 3.0. Pokud se na tyto změny podíváte, zjistíte, že těch hlavních nebylo v jednotlivých verzích nikdy moc, což se ovšem s verzí 3.0 změnilo, neboť ta nám přinesla řadu novinek, které mohou být pro programátory velmi užitečné. Jelikož jsou servlety klíčovou technologií JavaEE, ne náhodou se na internetu, již od vydání veřejného návrhu na finální verzi specifikace, který vyšel zhruba rok před ní, objevilo mnoho diskusí, které jednotlivé novinky (někdy i velmi divoce) probíraly a čas od času také kriticky hodnotily[9]. Převážně vlastnost týkající se automatické konfigurace webových frameworků (pomocí modularizace *deployment descriptoru*, anotací a programové definice servletů, filtrů a posluchačů), které použijeme pro realizaci našich servletových webových aplikací, nebyla přijímána zrovna vřele. Avšak již v následující pracovní verzi (jenž vyšla půl roku po veřejném návrhu), byly všechny hlavní nedostatky vyřešeny[10].

V předešlé kapitole, která pojednává o principech servletů, jsem záměrně nevyužil a nepopisoval žádnou vlastnost nové specifikace, jelikož všechny hlavní novinky, která verze 3.0 obsahuje, podrobněji popisuji až v následujících podkapitolách, kde každá podkapitola je věnována jedné konkrétní novince.

Informace o jednotlivých novinkách jsem získal jak ze samotné servletové specifikace, tak například z [3, 4].

3.1 Podpora asynchronního zpracování klientských požadavků

Jedna z největších novinek poslední verze servletů je zcela jistě možnost zpracovat uživatelské požadavky asynchronně. Avšak je nutné říci, že tuto vlastnost využijí převážně vývojáři frameworků, jelikož pro samotné programátory webových aplikací nepřináší žádnou přidanou hodnotu. Tato novinka vznikla hlavně pro účely zlepšení kvality služeb servletových kontejnerů při použití AJAX aplikací nebo obecně libovolných zdrojů, na jejichž odpovědi musí daná webová aplikace dlouho čekat. Takovéto zdroje mohou být například pomalé webové služby. Abych ale byl schopen vysvětlit princip, jakým je lepší kvalita služeb, pomocí asynchronního zpracování požadavků, docílena, je třeba také popsat způsob, jakým současné servletové kontejnery (respektive webové servery) zpracovávají požadavky klientů.

3.1.1 Pro každé spojení jedno vlákno

Až donedávna byl nejčastější způsob takový, že pro každé nové HTTP spojení, kontejner buď vytvořil nové, nebo získal z poolu volné, vlákno, kterému následně dal na starost nové spojení obsluhovat. Tedy kolik HTTP požadavků bylo v daný okamžik zpracovávaných, tolik činných vláken (pro obsluhu požadavků) server aktuálně obsahoval.

[9] Takovouto diskusi lze nalézt například zde: http://www.infoq.com/news/2008/12/servlet3_debate.
[10] Viz URL http://blogs.webtide.com/gregw/entry/servlet_3_0_proposed_final.

Důvod, proč většina moderních webových serverů tento způsob již nevyužívá, jsou především HTTP 1.1 keep-alive spojení. Dnešní prohlížeče totiž HTTP (respektive TCP) spojení po každé HTTP odpovědi neuzavírají, ale drží jej otevřené pro účely následných požadavků na tentýž webový server. Vlákno tedy může být jedním spojením blokováno velmi dlouho. A to i v případě, že je nečinné, jelikož klient aktuálně nezasílá žádné nové požadavky. Pokud navíc server využívá pool vláken, nebo již není schopen vytvořit vlákna nové, mohou být, vlivem blokace aktuálních vláken a tedy nedostatku vláken volných, odmítnuty požadavky ostatních klientů.

3.1.2 Co požadavek to vlákno

Moderní webové servery, výše zmíněný, problém řeší pomocí asynchronního API pro práci se „sokety“. Toto API tedy poskytuje takové operace, které neblokují volajícího. Lze se například jen dotázat, zda daný „soket“ náhodou neobsahuje nějaké nové data. Tímto způsobem je tedy server schopen všechny spojení obsluhovat jedním vláknem, a až v případě nových dat (například HTTP požadavku), získat z poolu další vlákno, které následně data zpracuje. Jakmile jsou data zpracována, je vlákno ihned vráceno do poolu nazpátek a je tak dostupné pro zpracování dalších údajů. Vlákna již tedy nejsou vytvářena pro každé nové spojení, ale pro každý nový požadavek. Tedy i v případě, že je spojení po dlouhou dobu otevřené a zároveň nečinné, není jím žádné vlákno blokováno. Server je tak schopen, jen velmi málo vlákny, obsloužit mnohonásobně vyšší počet uživatelů[11].

Ovšem s příchodem Web 2.0 aplikací, respektive s *XMLHttpRequest* objektem, má i tento přístup nedostatky. Web 2.0 aplikace totiž obvykle generují mnoho požadavků, na které server musí uvolňovat stejný počet vláken. Jelikož je však počet vláken omezen (servery vesměs využívají pool vláken s maximální kapacitou), může se stát, že pomocí nich nebude možné všechny požadavky obsloužit. Je tedy nutné, aby každý požadavek byl obsloužen v co nejkratší době, respektive, aby vlákno, které je serverem používáno pro obsluhu požadavků, bylo ihned opět v poolu. Bohužel AJAX požadavky většinou krátké nejsou, neboť mnohdy využívají různé webové služby, databázi či čekají na asynchronní událost[12]. Dostáváme se tak k příčině potřeby asynchronního zpracování požadavků, tedy k potřebě, aby požadavky mohly být zpracovány asynchronně na aplikačních vláknech, a aby vlákna, která kontejner využívá ke zpracování požadavků, nebyla po delší dobu blokována.

[11] O tomto faktu se můžete přesvědčit například z URL http://www.jboss.org/netty/performance.html.

[12] Čekání na asynchronní události je typické při využití comet principů, respektive server push technologie. Server push technologie znamená, že nejenom klient, ale také server je schopen začít HTTP komunikaci, tedy zaslat klientovi data, bez toho, aby je klient explicitně požadoval. Nicméně HTTP protokol je synchronní, a tak se využívá tzv. „long-polling“ techniky, která v tomto synchronním prostředí implementuje asynchronní přenos. Technika long-polling není nic jiného, než otevřené spojení na server, na kterém byl zaslán požadavek pro reakci na určitou událost. Klíčové je, že odpověď, na tento požadavek, není serverem zaslána ihned, ale až na základě výskytu konkrétní události. Jakmile je odpověď klientem obdržena, hned v zápětí je na server odeslán další požadavek, aby byl klient schopen reagovat také na další výskyt této události. Takhle se to opakuje pořád dokola. Zde můžete vidět, že zpracování požadavků, při využití long-polling techniky, může trvat velmi dlouho a je tedy potřeba, aby nebylo blokováno serverové vlákno.

3.1.3 API pro asynchronní zpracování požadavků

Jelikož Servlet 3.0 technologie byla vydána až 10. prosince 2009, kdy již samozřejmě existovalo spoustu Web 2.0 aplikací, jednotliví výrobci servletových kontejnerů realizovali asynchronní zpracování požadavků vlastními silami. Možná znáte třídy jako *Continuation*, *CometProcessor*, *AbstractAsyncServlet* či *CometEngine*. Každá z těchto tříd je totiž proprietární implementací asynchronního zpracování požadavků.

Nicméně dnes již servlety asynchronní zpracování požadavků podporují, proto postupně všichni výrobci přecházejí na toto standardizované API. Jen pro zajímavost, také API na klientské straně bude, s příchodem HTML 5[13] a tzv. Web Sockets[14], standardizováno. Tedy comet princip bude postupně nahrazen novou technologií.

A teď již k samotnému API. Jeho alfou a omegou je nové rozhraní *javax.servlet.AsyncContext* a metoda *startAsync()* v *ServletRequest* rozhraní. Dále je to příznak *asyncSupported*, který, v případě, že chceme asynchronní API využít, musí být pro daný servlet (popřípadě filtr) nastaven na hodnotu *true*. Lze jej nastavit jak v *deployment descriptoru*, tak programově nebo anotacemi. Programová registrace a anotace jsou probrány později.

Kdykoliv požadujete, aby byl požadavek zpracován asynchronně, musíte v metodě *service* (respektive *doXXX* metodách) zavolat metodu *startAsync* na přijatém *ServletRequest* objektu. Tato metoda Vám vrátí implementaci rozhraní *AsyncContext*, přes které je asynchronní zpracovávání řízeno, a lze ji v jednom asynchronním cyklu (tj. době, kdy je metoda *service* vykonávána) volat pouze jednou. Asynchronních cyklů totiž může být, pro jeden požadavek, více. Později tento fakt vysvětlím.

A v čem asynchronní zpracování vůbec spočívá? V případě kontejneru to je velmi jednoduché. Při tradičním zpracování požadavků je, po skončení metody *service*, odpověď ihned odeslána klientovi a požadavek je kontejnerem brán jako zpracovaný. Ovšem v případě asynchronního zpracování (tj. zavolání metody *startAsync* v aktuálním asynchronním cyklu), odpověď, ani po ukončení metody *service*, klientovi odeslána není. Toto je vše, o co se, při asynchronním zpracování, stará servletový kontejner. Tedy o to, zda má odpověď klientovi odeslat, či nikoliv a zda má tedy požadavek chápat jako zpracovaný. O vše ostatní, jako řízení asynchronního zpracování a odeslání odpovědi klientovi, se musí programátor daného servletu postarat sám.

Již jsem naznačil, že řízení asynchronního zpracování je prováděno přes *AsyncContext* objekt, který je získán metodou *startAsync*. Jak takovéto řízení může vypadat, je nejlepší vysvětlit na příkladu. Představte si situaci, že v určitém servletu potřebujete získat data z velmi pomalé webové služby, která následně vracíte jako odpověď pro klienta. Tuto situaci lze realizovat dvěma způsoby, a to synchronně a asynchronně. Synchronní způsob je velmi jednoduchý a spočívá v tom, že webová služba bude volána přímo v metodě *service* a bude se v ní tedy čekat na její odpověď. Poté, co se odpověď obdrží, se ihned

[13] Připravovaná HTML 5 specifikace je dostupná pod touto URL: http://www.w3.org/TR/html5/.

[14] Web Sockets technologie je modlou všech AJAX vývojářů, neboť přináší tolik očekávanou obousměrnou komunikaci prohlížeče a webového serveru. Je tedy kompletní náhradou všech dosavadních server push technik. Aktuální verzi specifikace můžete nalézt zde: http://dev.w3.org/html5/websockets/. Ale pozor, jedná se jen o definici DOM API, o protokolu, který se nakonec bude při komunikaci využívat, se vedou spory. Zajímavý článek na toto téma je dostupný pod URL http://blogs.webtide.com/gregw/entry/websockets_ietf_v_whatwg.

zašle klientovi. Jak je vidět, tento způsob je pro programátory velmi jednoduchý. Tato jednoduchost má však svou nevýhodu, a to zhoršenou kvalitou služeb použitého kontejneru, neboť je blokováno vlákno, které aktuální *service* metodu vykonává. Kontejner jej tedy není schopen vrátit zpět do poolu, a tak není dostupné pro obsluhu dalších požadavků. Pokud chceme kontejneru pomoci, využijeme asynchronního přístupu, jehož nevýhodou je ovšem o něco vyšší složitost implementace[15]. Asynchronní přístup spočívá v tom, že v metodě *service* webovou službu volat nebudeme, ale zavoláme ji až v jiném vlákně, na jehož skončení není třeba v metodě *service* čekat. V této metodě namísto toho zavoláme metodu *startAsync* a získáme tak *AsyncContext* objekt, dále vytvoříme vlastní vlákno, kterému *AsyncContext* objekt předáme, a následně vytvořené vlákno spustíme. Po spuštění vlákna, metoda *service* skončí a vlákno, které ji vykonávalo, bude schopné obsloužit další požadavky. Tím, že byla v metodě *service* volána metoda *startAsync*, doposud vytvořená odpověď (v našem případě prázdná) není kontejnerem odeslána klientovi. A co provede námi vytvořené vlákno? Zavolá webovou službu, získá její odpověď a zašle ji klientovi. To provede voláním metody *getResponse()* na obdrženém *AsyncContext* objektu, zapsáním načtených dat a zavoláním metody *complete()*, která zapříčiní to, že kontejner zašle zapsanou odpověď klientovi. Takovouto jednoduchou asynchronní implementaci obsahuje výpis 3. Další příklad využití asynchronního API můžete nalézt na přiloženém CD. Jedná se o realizaci jednoduché „chatovací“ aplikace.

```
@WebServlet(urlPatterns = "/AsyncServlet", asyncSupported = true)
public class AsyncServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        AsyncContext aCtx = request.startAsync();
        new Thread(new AsyncRunnable(aCtx)).start();
    }
}

public class AsyncRunnable implements Runnable {

    private final  AsyncContext ctx;

    public AsyncRunnable(AsyncContext ctx) {
        this.ctx = ctx;
    }

    public void run() {
        try {
            ServletResponse response = ctx.getResponse();
            PrintWriter  out = ctx.getResponse().getWriter();
```

[15]Vyšší složitost asynchronního přístupu nemusí každému vyhovovat. Pěknou polemiku o tomto problému můžete nalézt například zde: http://blog.krecan.net/2010/01/28/proc-nepotrebuji-asynchronni-jdbc/. Tento článek navíc nabízí odkaz na zajímavou prezentaci, která obhajuje starší model vlákna pro spojení, a tedy nepřímo říká, že vlastně žádné asynchronní zpracovávání požadavků není potřeba.

```
                // volani webove sluzby
                // zapsani odpovedi pro klienta
                // out.print() ...
            } catch (Exception ex) {
                // reakce na chybu
            } finally {
                // odeslani odpovedi klientovi
                ctx.complete();
            }
        }
    }
```

Výpis 3: Ukázková implementace asynchronního zpracování požadavku

3.1.3.1 Další části asynchronního API Asynchronní API není jen o metodách *startAsync* a *complete*, i když je pravdou, že jsou jeho hlavním pilířem.

Zvláště rozhraní *AsyncContext* obsahuje další užitečné metody:

- Metody *getRequest()* a *getResponse()*. Pomocí těchto metod lze získat požadavek a odpověď, které obdržela daná *service* metoda, ve které byl *AsyncContext* vytvořen, nebo objekty, které byly předány *startAsync* metodě, jelikož ta je dostupná i ve variantně, která přijímá *ServletRequest* a *ServletResponse* objekty.

- Metody *dispatch()*, *dispatch(String)* a *dispatch(ServletContext, String)*. Tyto metody jsou velmi užitečné a jsou alternativou k metodě *complete*. Ony vlastně tvoří nový způsob asynchronního zpracování požadavků, kdy odpověď není odeslána klientovi v asynchronním vlákně, ale opět v metodě *service*. Dispatch metody fakticky provádějí *RequestDispatcher.include(ServletRequest, ServletResponse)* s požadavkem a odpovědí, které získají pomocí metod *getRequest*, respektive *getResponse*. Zásadní ale je, že zahajují nový asynchronní cyklus, tedy další volání metody *service* pro stejný požadavek. Klíčový je také fakt, že každý asynchronní cyklus opět aktivuje synchronní zpracování daného požadavku. Tedy pokud není v následující metodě *service* znovu vytvořen *AsyncContext* objekt (který je pro všechny asynchronní cykly stejný, jelikož je vázán k požadavku), je po jejím vykonání požadavek chápán jako zpracovaný a vytvořená odpověď odeslána klientovi. Pokud by však *AsyncContext* vytvořen byl, mohl by se zahájit další asynchronní cyklus anebo by se mohla zavolat metoda *complete*.

- Metody *setTimeout(long)* a *getTimeout()*. Tyto metody slouží k nastavení doby, jenž vyjadřuje maximální čas, do kterého musí být zavolána metoda *complete*, popřípadě jedna z metod *dispatch*, na tomto *AsyncContext* objektu. Pokud se tak nestane, kontejner notifikuje všechny posluchače, a pokud ani ty metodu *complete* nezavolají, zavolá jí sám a ukončí tak zpracovávání požadavku.

- Metoda *addListener()*. Umožňuje přidat posluchače pro tento *AsyncContext* objekt. Posluchači mohou reagovat na zavolání metody *complete*, na vypršení limitu, na chybu nebo na start nového asynchronního cyklu.

Také *ServletRequest* rozhraní obsahuje několik dalších pomocných metod:

- Metoda *isAsyncSupported()*. Tato metoda vrací booleovskou hodnotu, která značí, zda aktuální servlet (či filtr) podporuje asynchronní zpracování požadavků. Tedy, zda nastavil svůj *asyncSupported* příznak na hodnotu *true*.
- Metoda *isAsyncStarted()*. Značí, zda je požadavek aktuálně zpracováván asynchronně.
- Metoda *getAsyncContext()*. Vrací *AsyncContext* objekt, který byl obdržen posledním voláním metody *startAsync*. Samozřejmě v době volání této metody, musí metoda *isAsyncStarted* vracet hodnotu *true*.
- Metoda *getDispatcherType()*. Získá typ odesílatele tohoto požadavku. Typem může být také hodnota *ASYNC*, která značí, že metoda *service* (popřípadě *doFilter*), byla zavolána pomocí *AsyncContext* objektu. Tuto metodu lze, s výhodou, využít při použití metody *dispatch* bez parametrů, jelikož je volán opět původní servlet.

3.2 Modularizace deployment descriptoru

Hlavní tvůrce nejnovější verze servletů Rajiv Mordani říká, že hlavní hybnou silou pro vytvoření nové specifikace byla především snaha docílit snazší konfigurace webových aplikací. Tímto bylo především myšleno to, že programátoři aplikací již nebudou muset provádět nudné nastavování použitých frameworků ve svých *deployment descriptorech* a budou moci přenechat tuto práci na samotné vývojáře daných frameworků.

Specifikace tuto požadovanou vlastnost řeší hned třemi způsoby, které vývojáři knihoven (ale i samotní programátoři webových aplikací) mohou libovolně použít. Prvním způsobem jsou tzv. „*fragmenty deployment descriptoru*“, které popisuji v této podkapitole, druhým, respektive třetím anotace pro deklaraci servletů, filtrů a posluchačů a jejich programová registrace. Druhý a třetí způsob jsou vysvětleny v následujících podkapitolách.

Jak již bylo řečeno, fragmenty *deployment descriptoru* byly vytvořeny z toho důvodu, aby samotný *web.xml* soubor nemusel obsahovat nastavení použitých knihoven a frameworků. Tento požadavek je právě fragmenty řešen, neboť to jsou vlastně „knihovní *deployment descriptory*“, které si jednotlivé knihovny nesou ve svých JAR souborech, a ve kterých provádí veškeré nastavení, které je potřeba k jejich bezproblémovému chodu. Toto nastavení již tedy nemusí být obsaženo v samotném *deployment descriptoru* webové aplikace. Jednotlivé fragmenty tedy můžeme chápat jako jakési části souboru *web.xml*, který je z nich, při nahrávání webové aplikace, servletovým kontejnerem nakonec složen. Způsob skládání je popsán dále v textu.

Přestože jsou fragmenty pouze části *deployment descriptoru*, jsou s ním téměř identické, respektive mohou obsahovat většinu jeho elementů a atributů. To v čem se fragmenty odlišují, není v jejich obsahu, ale v počtu, názvu a umístění. Každý fragment musí být vždy nazván *web-fragment.xml*, musí být umístěn ve složce *META-INF* JAR souboru konkrétní knihovny ve *WEB-INF/lib* a jeho kořenový element musí být namísto *web-app* pojmenován *web-fragment*. Jelikož aplikace většinou obsahuje více knihoven, je povolen i vícenásobný počet fragmentů. Na výpisu 4 můžete vidět ukázku fragmentu, který definuje jeden servlet a jeden posluchač a je pojmenován „*Fragment1*“.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-fragment version="3.0" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3
    .org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http:
    //java.sun.com/xml/ns/javaee/web-fragment_3_0.xsd">
    <name>Fragment1</name>
    <servlet>
        <servlet-name>HelloWorldServlet</servlet-name>
        <servlet-class>
            HelloWorldServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloWorldServlet</servlet-name>
        <url-pattern>/HelloWorldServlet</url-pattern>
    </servlet-mapping>
    <listener>
        <listener-class>
            HelloWorldListener
        </listener-class>
    </listener>
</web-fragment>
```

Výpis 4: Jednoduchý fragment *deployment descriptoru*

Poněvadž aplikace nezřídka kdy obsahují také knihovny, které úplně neznají, obsahuje deployment descriptor atribut *metadata-complete*, který v případě, že je nastaven na hodnotu *true*, nařizuje servletovému kontejneru ignorovat všechny *web-fragment.xml* soubory, které nalezne v jednotlivých knihovnách ve složce *WEB-INF/lib* konkrétní webové aplikace. Tedy aplikace tímto vyjadřuje, že veškeré nastavení pro její správnou funkčnost je již uvedeno v souboru *web.xml*, a že žádné další nastavení již není potřeba. Atribut *metadata-complete* je definován v kořenovém *web-app* elementu.

3.2.1 Řazení fragmentů

Pořadí, v jakém jsou jednotlivé fragmenty kontejnerem zpracovávány (tedy je tvořen výsledný *deployment descriptor*, přičemž platí, že shodné nastavení nemůže být následnými fragmenty již přepsáno), je implicitně náhodné. Pokud je však pro danou webovou aplikaci signifikantní, lze jednotlivé fragmenty seřadit a zajistit tak požadované pořadí jejich zpracování.

Řazení je dvojího druhu, absolutní a relativní, protože se může stát, že nejen webová aplikace, ale i samotné knihovny potřebují řazení specifikovat. Relativní řazení tedy mohou provádět pouze knihovny, tj. může být provedeno pouze v jednotlivých fragmentech, absolutní se pak může objevit výhradně v souboru *web.xml*. Platí, že pokud je absolutní řazení definováno, veškeré relativní řazení jsou ignorována.

Aby však řazení mohlo být vůbec provedeno, fragmenty mohou obsahovat element *name*, který slouží k jejich pojmenování a jehož hodnota je ve vlastním řazení využívána. Tento element byl použit i ve výše zmíněné ukázce fragmentu, který byl nazván „*Frag-*

ment1“. Nicméně element *name* je nepovinný, jelikož, jak dále uvidíte, řazení počítá i s nepojmenovanými fragmenty.

3.2.1.1 Absolutní řazení Absolutní řazení je specifikováno novým *web.xml* elementem *absolute-ordering*. Tento element může dále obsahovat libovolný počet elementů *name* a maximálně jeden element *others*, který je, mezi elementy *name*, možno uvést na libovolné pozici.

Význam těchto elementů je následující. Jednotlivé elementy *name* korespondují s *name* elementy ve fragmentech. Tedy pokud je jejich hodnota stejná, element *name* v *absolute-ordering* elementu vlastně vyjadřuje daný fragment. Kdežto element *others*, jak zřejmě tušíte, zastupuje všechny fragmenty (včetně těch nepojmenovaných), které nejsou explicitně jednotlivými *name* elementy uvedeny. Jednotlivé fragmenty, které servletový kontejner v aplikaci nalezne, jsou pak zpracovány přesně v takovém pořadí, které je uvedeno v elementu *absolute-ordering*. Relativní řazení není bráno v potaz, a to ani v případě použití elementu *others*. Dále platí, že absence *others* elementu způsobí ignorování všech fragmentů, které nebyly pomocí name elementů uvedeny.

Příklad 3.1
Mějme pět fragmentů, kde jeden je nepojmenován, a zbylé čtyři se nazývají *A*, *B*, *C* a *D*. Pokud by element *absolute-ordering* vypadal následovně,

```
<absolute−ordering>
    <others/>
    <name>D</name>
    <name>C</name>
    <name>A</name>
</absolute−ordering>
```

fragmenty by se zpracovaly v jednom z následujících pořadí: *nepojmenovaný*, *B*, *D*, *C*, *A* nebo *B*, *nepojmenovaný*, *D*, *C*, *A*.

Pokud by element *absolute-ordering* obsahoval toto,

```
<absolute−ordering>
    <name>B</name>
    <name>E</name>
</absolute−ordering>
```

fragmenty by byly zpracovány následovně: *B*, *E*. Tedy všechny fragmenty, mimo fragmentů *B* a *E*, by kontejner ignoroval. ■

3.2.1.2 Relativní řazení Relativní řazení smí definovat pouze jednotlivé fragmenty a je aktivní jen tehdy, pokud *deployment descriptor* neobsahuje element *absolute-ordering*.

Pro relativní řazení se používá nového elementu *ordering*, který obsahuje maximálně jeden element *after* a maximálně jeden element *before*. Tyto dva elementy dále mohou obsahovat libovolný počet elementů *name*, následovaných maximálně jedním elementem *others*.

Již z názvů těchto elementů vyplývá jejich sémantika. Elementem *after*, respektive *before* může daný fragment specifikovat, že je nutné jej zpracovat po, respektive před určitým fragmentem. Pokud je navíc v těchto elementech uveden také element *others*, je daný fragment zpracován po všech fragmentech, respektive před všemi fragmenty, kteří element *others* v *after*, respektive *before* elementu neuvedli. Je zřejmé, že element *others* je možné deklarovat pouze v jednom z elementů *after* a *before*. Nelze jej přidat do obou těchto elementů najednou. Dále platí, že i v případě použití tohoto elementu, je stále platné i ostatní specifikované pořadí, neboť element *others* můžou, ve stejných elementech, využít také ostatní fragmenty.

Je jasné, že pomocí relativního řazení, lze snadno dosáhnout cyklických závislostí. Pokud se takovéto závislosti objeví, kontejner je povinen nahlásit chybu a ukončit nahrávání dané webové aplikace.

Příklad 3.2
Mějme webovou aplikaci, která obsahuje tři fragmenty s následujícím obsahem (absolutní řazení není specifikováno):

```
obsah prvniho fragmentu
 <web−fragment>
     <name>A</name>
     ...
     <ordering>
         <after><name>B</name></after>
     </ordering>
     ...
 </web−fragment>

 obsah druheho fragmentu
 <web−fragment>
     <name>B</name>
     ...
 </web−fragment>

 obsah tretiho  fragmentu
 <web−fragment>
     <name>C</name>
     ...
     <ordering>
         <before><others/></before>
     </ordering>
     ...
 </web−fragment>
```

V tomto případě se fragmenty mohou zpracovat pouze v pořadí *C*, *B*, *A*. Další příklady relativního řazení můžete nalézt ve specifikaci v kapitole 8.2.2. ■

3.2.2 Vytváření deployment descriptoru z fragmentů

Jak dále uvidíte, servlety, filtry a posluchače je nově možné také deklarovat pomocí anotací. Z těchto důvodů, nejnovější verze servletové specifikace připouští absenci *deployment descriptoru*, tj. souboru *web.xml*.

Pokud webová aplikace tento soubor neobsahuje, nic se neděje, jelikož je kontejnerem i tak, na základě anotací a obsahu fragmentů, vytvořen. Pokud jej však obsahuje, je soubor *web.xml* o tyto data pouze rozšířen.

Následující výpis obsahuje ty nejhlavnější pravidla, která jsou při rozšiřování (popřípadě vytváření) deployment descriptoru platná.

1. Informace uvedené v souboru *web.xml* mají nejvyšší prioritu. Tedy v případě konfliktních (překrývajících se) nastavení, kontejner akceptuje pouze data ze souboru *web.xml*. Ostatní údaje jsou zcela nebo částečně ignorovány (záleží na typu konkrétních dat).

2. Pokud *web.xml* soubor obsahuje element *metadata-complete* nastavený na *true*, jsou všechny fragmenty a anotace ignorovány. Soubor *web.xml* již není o žádné data rozšířen.

3. S daty z fragmentů je zacházeno stejně, jako s daty ze souboru *web.xml*.

4. Pořadí, v jakém jsou data z fragmentů do souboru *web.xml* přidávána, je určeno absolutním nebo relativním řazením.

5. Zpracování anotací dané knihovny je prováděno až po integraci knihovního *web-fragment.xml* souboru, ale před zpracováním následujícího fragmentu.

6. Pokud je ve fragmentu nastaven atribut *metadata-complete* na *true*, anotace nejsou v dané knihovně zpracovávány.

7. Nastavení, které je uvedeno v souborech *web-fragment.xml*, má vyšší prioritu než nastavení pomocí anotací.

8. Elementy, které se mohou opakovat více než jednou, jsou aditivní.

9. Kdykoliv dva různé fragmenty obsahují konfliktní nastavení, přičemž toto nastavení není uvedeno ani ve *web.xml*, musí kontejner přerušit nahrávání webové aplikace a oznámit chybu.

Toto byl výčet těch nejdůležitějších pravidel, které každý kontejner musí dodržovat. Vyčerpávající popis všech pravidel je pak uveden v kapitole 8.2.3 servletové specifikace. Tato kapitola taktéž obsahuje přesnou definici jednotlivých konfliktních situací, jenž mohou při skládání *web.xml* souboru nastat.

3.3 Definice servletů, filtrů a posluchačů pomocí anotací

Tvůrcům JSR 315 zřejmě „zeštíhlení“ *deployment descriptoru* pomocí fragmentů nestačilo, jelikož vytvořili další velkou změnu, kterou jsou anotace *WebServlet*, *WebFilter* a *WebListener* (všechny tři náleží do nového balíčku *javax.servlet.annotation*). Tyto anotace nahrazují část funkcionality souboru *web.xml*, respektive *web-fragment.xml* a činí jej nepovinným.

Anotace tedy umožňují programátorům definovat servlety, filtry a posluchače a jsou plnohodnotnou alternativou k *servlet*, *servlet-mapping*, *filter*, *filter-mapping* a *listener* elementům *deployment descriptoru*.

Důležitou vlastností těchto anotací je, že je lze použít i v JAR souborech ve složce *WEB-INF/lib* a je respektován, výše popsaný, *absolute-ordering* element. Tedy v případě, že je daná knihovna tímto elementem vyloučena, také všechny její servlety, filtry a posluchače, které deklaruje pomocí anotací, jsou ignorovány. Zpracování anotací lze však potlačit ještě jedním způsobem, a to nastavením atributu *metadata-complete* na *true*. Ovšem pokud tento element na *true* nastavíte, budou ignorovány nejen anotace v knihovnách, ale také anotace na třídách ve *WEB-INF/classes*.

Anotace *WebServlet* slouží k definování servletů a můžeme pomocí ní nastavit informace jako URL mapování, název, popis, pořadí nahrávání při startu servletového kontejneru, inicializační parametry, podporu asynchronního zpracování požadavků a ikonky. Tuto anotaci lze využít pouze pro HTTP servlety, tedy pro všechny třídy, které rozšiřují abstraktní třídu *HttpServlet*. Na výpisu 5 můžete vidět ukázku použití této anotace.

```
@WebServlet(name = "HelloWorld", urlPatterns = {"/helloworld"})
public class HelloWorldServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
    }
}
```

Výpis 5: Použití anotace *WebServlet*

Pomocí anotace *WebFilter* jsme schopni deklarovat filtry a je nám umožněno nastavit například jejich URL mapování (popřípadě jména servletů, ke kterým bude filtr patřit), inicializační parametry, druhy požadavků, při kterých se filtr aktivuje a podporu asynchronního módu. Výpis 6 dává příklad, jak lze tuto anotaci použít.

```
@WebFilter(filterName = "MockFilter", urlPatterns = {"/*"})
public class MockFilter implements Filter {

    private FilterConfig filterConfig = null;

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain)
            throws IOException, ServletException {
        chain.doFilter(request, response);
    }
```

```
    public void init(FilterConfig filterConfig) {
        this.filterConfig = filterConfig;
    }

    public void destroy() {
    }

    public FilterConfig getFilterConfig() {
        return (this.filterConfig);
    }
}
```

Výpis 6: Použití anotace *WebFilter*

Poslední anotací je anotace *WebListener*, která nám umožňuje deklarovat různé typy posluchačů ve webové aplikaci. Lze ji použít pro definici posluchačů jako *ServletContextListener*, *ServletContextAttributeListener*, *ServletRequestListener*, *ServletRequestAttributeListener*, *HttpSessionListener* a *HttpSessionAttributeListener*. Jediným parametrem této anotace je řetězec, který vyjadřuje popis daného posluchače. Na výpisu 7 můžete vidět definici kontextového posluchače pomocí anotace *WebListener*.

```
@WebListener
public class ContextListener implements ServletContextListener {

    public void contextInitialized(ServletContextEvent sce) {
    }

    public void contextDestroyed(ServletContextEvent sce) {
    }
}
```

Výpis 7: Použití anotace *WebListener*

3.4 Programová registrace servletů, filtrů a posluchačů

Další novinkou, která má především usnadnit práci programátorům frameworků pro tvorbu webových aplikací, je programová registrace servletů, filtrů a posluchačů. Tato možnost je tedy po *deployment descriptorech* (či jejich fragmentech) a anotacích třetím způsobem, jak lze servlety, filtry a posluchače do webových aplikací přidat. Programátoři frameworků mohou tímto způsobem například registrovat své řídící servlety.

Programovou registraci je však možné použít pouze při inicializaci dané webové aplikace, tedy v posluchačích *ServletContextListener*, respektive *ServletContainerInitializer*. Rozhraní *ServletContainerInitializer* je další novinkou v nejnovější specifikaci a popisuji jej později.

Programová registrace servletů se od programové registrace filtrů, respektive posluchačů příliš neliší, proto zde dále popíši pouze ji.

Servlet můžeme programově přidat jednou ze tří přetěžených metod v *ServletContext* rozhraní. Jsou to nové metody s názvem *addServlet*, kde každá přijímá jednak unikátní jméno servletu, tak jak ho známe z *deployment descriptoru*, tak daný servlet v určité podobě. První jej přijímá jako řetězec s plně kvalifikovaným názvem jeho třídy, druhá jako objekt typu *Class* a třetí přijímá rovnou jeho instanci. Každá z těchto metod vrací implementaci nového rozhraní *javax.servlet.ServletRegistration.Dynamic*, které nám dovoluje dodatečnou konfiguraci registrovaného servletu. Můžeme například nastavit jeho URL mapování, inicializační parametry, bezpečnostní omezení apod. Další důležitou vlastností je, že tyto metody (kromě té, která servlet přijímá rovnou jako instanci) respektují anotace, které jsou na předaných třídách specifikovány. Specifikace nařizuje, že anotace jako *RunAs*, *DeclareRoles*, *ServletSecurity* a *MultipartConfig* musí být kontejnery brány v potaz (Pozor! Anotace *WebServlet* je ignorována.). Tímto je programátor, servletů pro registraci, alespoň částečně ušetřen od potřeby jejich další konfigurace přes obdržené *ServletRegistration.Dynamic* objekty. Anotace *ServletSecurity* a *MultipartConfig* jsou nové a taktéž je popisuji v následujících částech této práce.

Metoda *addServlet(String, Servlet)*, která daný servlet přijímá jako jeho instanci, je v API z toho důvodu, že daný servlet může být ještě před svou registrací konfigurován. Totiž servlety, které již jednou registrujeme, dále není možné přes rozhraní *ServletRegistration.Dynamic* nastavovat, jelikož metody *addServlet* pro již registrované servlety vrátí hodnotu *null*. *Null* hodnota je vrácena i pro servlety definované pomocí anotací, respektive *deployment descriptoru*. Pro snadné vytvoření instancí servletů pro registraci, API obsahuje další pomocnou metodu *createServlet(Class)*, která z obdržené třídy servletu vytvoří jeho instanci. Tato metoda taktéž respektuje anotace a, stejně jako *addServlet* metody, také automatickou inicializaci závislostí dle nového JSR 299[16].

Krom výše zmíněných metod, API dále obsahuje metody *getServletRegistration(String)*, respektive *getServletRegistrations()*. Tyto metody při zavolání vrací objekt, respektive mapu objektů typu *javax.servlet.ServletRegistration*. Toto rozhraní je předkem rozhraní *ServletRegistration.Dynamic* a můžeme pomocí něj taktéž nastavit různé informace pro konkrétní servlet. Zásadní změnou ale je, že *ServletRegistration* objekty lze získat opakovaně a kontejner je vrací nejen pro programově registrované servlety, ale pro všechny servlety z daného kontextu. Tedy i ty, které byly deklarovány pomocí *deployment descriptoru* nebo anotací. Avšak přes *ServletRegistration* rozhraní lze měnit jen URL mapování a inicializační parametry. Ostatní údaje, jako například bezpečnostní omezení, konfigurovat nelze.

Na výpisu 8 můžete vidět implementaci *ServletContextListener* posluchače, který programově registruje servlet, nastavuje jeho URL mapování a další informace a poté vypisuje všechny servlety, které webová aplikace aktuálně obsahuje.

```
@WebListener
public class ContextListener implements ServletContextListener {

    public void contextInitialized (ServletContextEvent sce) {
```

[16]JSR 299, krom jiného, rozšiřuje koncept tzv. Managed Bean a do JavaEE přidává IoC technologii.

```
        try {
            // vytvoreni instance naseho servletu
            HelloWorldServlet hWorldServ = sce.getServletContext().createServlet(
                HelloWorldServlet.class);

            // zde muze byt pripadna konfigurace vytvoreneho hWorldServ servletu

            // registrace servletu
            ServletRegistration.Dynamic hWorldServReg = sce.getServletContext().addServlet("
                HelloWorldServlet", hWorldServ);

            // nastaveni parametru pro servlet
            hWorldServReg.addMapping("/helloworld");
            hWorldServReg.setLoadOnStartup(0);

            // vypis vsech servletu webove aplikace
            Map<String, ? extends ServletRegistration> webAppServlets = sce.getServletContext
                ().getServletRegistrations();
            for (ServletRegistration reg : webAppServlets.values()) {
                sce.getServletContext().log(reg.getName() + ":" + reg.getClassName());
            }
        } catch (ServletException servException) {
            // reakce na chybu pri registrovani servletu
        }
    }

    public void contextDestroyed(ServletContextEvent sce) {
    }
}
```

Výpis 8: Programová registrace servletu

3.5 Sdílení zdrojů v JAR souborech

To, že knihovny nově mohou, ať již pomocí fragmentů nebo programové registrace, definovat servlety, filtry či posluchače, již víme. Ony však mohou webovou aplikaci obohatit i o klasické HTML stránky, respektive libovolné statické zdroje[17]. Stačí, aby tyto soubory umístili do speciální složky svého JAR souboru.

Tato složka se musí jmenovat *resources* a musí být umístěna ve složce *META-INF*. Všechny soubory, které do této složky knihovna umístí, budou přímo dostupné všem uživatelům dané webové aplikace. Tedy bude pro ně vytvořeno konkrétní URL. Princip mapování těchto souborů na URL je velmi jednoduchý, neboť je naprosto stejný, jako kdyby složka *resources* byla složkou *kontextovou*. Samozřejmě v případě shodnosti názvů, má přednost ten soubor, který je umístěn ve složce *kontextové*.

Soubory v knihovnách lze také načíst pomocí *ServletContext* metod *getResource(String)*, respektive *getResourceAsStream(String)*, kde před knihovnami je samozřejmě prohledávána nejprve *kontextová* složka. Také je dobré vědět, že pořadí prohledávání knihoven není

[17] Je dovoleno přidat také JSP stránky.

nijak specifikováno, tedy je závislé na implementaci aktuálního servletového kontejneru, a že element *absolute-ordering* nemá na knihovní statické zdroje žádný vliv. Tedy, i když danou knihovnu pomocí tohoto elementu vyřadíme, stejně se její *META-INF/resources* složka bude prohledávat.

Příklad 3.3
Jak sdílení zdrojů funguje, je nejlepší ukázat na příkladu. Mějme následující webovou aplikaci, která je dostupná pod URL *http://localhost:8080/app1*:

/index.html
/WEB-INF/lib/Knihovna1.jar!/META-INF/resources/static/obr.jpg
/WEB-INF/lib/Knihovna1.jar!/META-INF/resources/index.html

Pokud bychom se dotazovali na URL ve tvaru *http://localhost:8080/app1/index.html*, byl by nám vrácen obsah souboru *index.html* z *kontextové* složky, neboť má před knihovním *index.html* souborem přednost. Pokud však požádáme o *http://localhost:8080/app1/static/obr.jpg*, servletový kontejner zjistí, že žádný takový soubor *kontextová* složka neobsahuje, a tak začne prohledávat všechny JAR soubory, dokud nenarazí na zdroj, který tomuto URL odpovídá. V našem případě takovýto soubor najde (v knihovně s názvem „*Knihovna1.jar*“), a tak vrátí jeho obsah. ■

3.6 Sdílení knihoven servletového kontejneru

Servletové kontejnery, respektive aplikační servery, většinou nabízejí mnoho knihoven, které realizují různé JavaEE nebo i jiné technologie. Pokud však tyto technologie chce daná webová aplikace využít, obvykle je nucena provést i jejich konfiguraci ve svém *deployment descriptoru*. Tato konfigurace je však pro každou webovou aplikaci většinou stejná, a tak si expertní skupina, která stojí za prozatím poslední verzí servletů, řekla, že by bylo dobré, aby se knihovny, které jsou v rámci daného kontejneru dostupné všem webovým aplikacím, mohly do těchto aplikací nakonfigurovat samy. Takovéto řešení tedy nakonec nejnovější specifikace přidává a je řešeno klasicky pomocí posluchačů.

Princip je takový, že servletový kontejner při startu nejprve zjistí všechny aplikace, které jsou na něm nahrány, a pak následně provede notifikaci jednotlivých knihoven, kterým postupně každou aplikaci, respektive její *ServletContext* objekt, předá. Tato knihovna se pak pomocí programového přidávání servletů, filtrů, či posluchačů, může do dané aplikace nakonfigurovat.

Notifikace je realizována rozhraním *javax.servlet.ServletContainerInitializer*, jehož implementaci musí každá knihovna, jenž chce být notifikována, poskytovat. Implementační třídy tohoto rozhraní jsou kontejnerem hledány standardním *ServiceLoader*[18] mechanismem, tedy jejich plně kvalifikované názvy musí být zapsány v souboru s názvem *javax.servlet.ServletContainerInitializer*, který je každá knihovna nucena umístit do složky *META-INF/services* svého JAR souboru.

[18] Více informací viz http://java.sun.com/developer/technicalArticles/javase/extensible/.

K rozhraní *ServletContainerInitializer* specifikace také zavádí novou anotaci *javax.servlet.annotation.HandlesTypes*, kterou může být konkrétní implementace tohoto rozhraní anotována. Tato anotace specifikuje datové typy, o které má implementace, tedy daná knihovna, zájem. Je na servletovém kontejneru, aby při startu každé webové aplikace prošel všechny její datové typy, a pokud je některý typ v konkrétní anotaci *HandlesTypes* uveden (popřípadě je uveden jeho rodič, rozhraní, které realizuje, nebo anotace, kterou je anotován), je při notifikaci implementace, ke které tato anotace patří, implementaci, ve formě objektu typu *Class*, předán.

Rozhraní *ServletContainerInitializer* obsahuje jedinou metodu *onStartup(Set<Class>, ServletContext)*, která tedy krom *ServletContext* objektu, přijímá také množinu objektů typu *java.lang.Class*. Tato množina je kontejnerem naplněna těmi typy, které byly pomocí anotace *HandlesTypes* požadovány, a jenž aktuální webová aplikace obsahuje. Pokud implementace nebyla anotací *HandlesTypes* anotována, nebo aktuální webová aplikace neobsahuje žádné typy, které byly požadovány, je namísto množiny vrácena hodnotu *null*. Tímto konkrétní implementace ihned zjistí, že aktuální webová aplikace její knihovnu nepotřebuje.

Pokud daná knihovna realizuje rozhraní *ServletContainerInitializer* a je zároveň umístěna ve složce *WEB-INF/lib*, tedy není dostupná všem aplikacím, nýbrž pouze jedné, je notifikována pouze při startu této webové aplikace. O inicializaci ostatních webových aplikací se nedozví. Pokud je navíc také vyřazena pomocí *absolute-ordering* elementu, její případné implementace typu *ServletContainerInitializer* jsou ignorovány. Taktéž je nutné zmínit, že metoda *onStartup* je volána před všemi posluchači konkrétní webové aplikace.

Pro inspiraci, jak může být konkrétní *ServletContainerInitializer* implementován, se, prosím, podívejte na výpis 9. Ten obsahuje realizaci zavaděče pro JSF 2.0[19], která již výhod rozhraní *ServletContainerInitializer* využívá.

```
@HandlesTypes({
    javax.faces.component.FacesComponent.class
    // ruzne dalsi JSF specificke typy
})
public class FacesInitializer implements ServletContainerInitializer {
    private static final String FACES_SERVLET_CLASS = javax.faces.webapp.FacesServlet.
        class.getName();

    public void onStartup(Set<Class<?>> classes, ServletContext servletContext)
            throws ServletException {

        if (shouldCheckMappings(classes, servletContext)) {
            Map<String, ? extends ServletRegistration> existing = servletContext.
                getServletRegistrations();
            for (ServletRegistration registration : existing.values()) {
                if (FACES_SERVLET_CLASS.equals(registration.getClassName())) {
                    // FacesServlet jiz byl definovan
                    return;
                }
```

[19] JSF 2.0 technologie je definována v JSR 314.

```
            }

            // zaregistrujeme FacesServlet
            ServletRegistration reg =
                    servletContext.addServlet("FacesServlet",
                    "javax.faces.webapp.FacesServlet");
            // namapujeme jej na potrebne URL
            reg.addMapping("/faces/*", "*.jsf", "*.faces");
        }
    }

    private boolean shouldCheckMappings(Set<Class<?>> classes,
            ServletContext context) {
        if (classes != null && !classes.isEmpty()) {
            // aktualni webova aplikace vyuziva JSF
            return true;
        }

        return false;
    }
}
```

Výpis 9: Implementace rozhraní *ServletContainerInitializer* v JSF 2.0

3.7 Anotace pro definici bezpečnostních omezení

Již víme, že servlety lze nyní deklarovat také pomocí anotace *WebServlet*. Tuto anotaci však již nelze použít pro definování jejich bezpečnostních omezení. Tento nedostatek specifikace tedy řeší tím, že zavádí tři další anotace, které můžeme k těmto účelům použít. Tyto anotace jsou tedy rovnocennou alternativou k *security-constraint* elementu *deployment descriptoru*.

Hlavní anotací je anotace *javax.servlet.annotation.ServletSecurity*, která se deklaruje na jednotlivé servletové třídy. Nelze ji tedy specifikovat přímo na určité *doXXX* metody, ale, jak dále uvidíte, není to ani potřeba. Zbylými anotacemi jsou anotace *HttpConstraint*, respektive *javax.servlet.annotation.HttpMethodConstraint*, které se ale používají pouze jako parametry při vytváření anotace *javax.servlet.annotation.ServletSecurity*.

Pomocí anotací *HttpConstraint* a *HttpMethodConstraint* se specifikují bezpečnostní omezení ke všem, respektive k jednotlivým HTTP metodám. U obou je tedy možné specifikovat jednak role, kterým je dovoleno dané HTTP metody využít, tak omezení na bezpečnost přenosu dat. Taktéž můžete definovat, zda prázdná množina rolí znamená úplnou nedostupnost, nebo pravý opak, tedy přístup všem uživatelům. U anotace *HttpMethodConstraint* musí být navíc uvedena také HTTP metoda, ke které se omezení vztahují.

Jak jsem již zmínil, anotace *HttpConstraint* a *HttpMethodConstraint* slouží pouze jako parametry při vytváření *ServletSecurity* anotace. Ta tedy přijímá pole objektů *HttpMethodConstraint* a jednu instanci typu *HttpConstraint*. Platí, že omezení specifikována objektem *HttpConstraint*, jsou vztažena na všechny metody, které nebyly explicitně uvedeny po-

mocí *HttpMethodConstraint* objektů. Také je vhodné říci, že pole *HttpMethodConstraint* je implicitně prázdné a pokud nezadáte ani objekt *HttpConstraint*, vytvoří se jeho výchozí hodnota, která povolí vstup všem uživatelům a nezabezpečený přenos dat.

Příklad 3.4
Pro lepší pochopení sémantiky bezpečnostních anotací, se, prosím, podívejte na tyto příklady, které jsou uvedeny přímo ve specifikaci:

- Žádná omezení pro všechny HTTP metody:

```
@ServletSecurity
public class Example1 extends HttpServlet {
}
```

- U všech metod musí být použit bezpečný přenos dat:

```
@ServletSecurity(@HttpConstraint(transportGuarantee = TransportGuarantee.CONFIDENTIAL))
public class Example2 extends HttpServlet {
}
```

- Zde je jakýkoliv přístup zamítnut:

```
@ServletSecurity(@HttpConstraint(EmptyRoleSemantic.DENY))
public class Example3 extends HttpServlet {
}
```

- Přístup ke všem HTTP metodám je povolen pouze uživatelům s rolí *R1*:

```
@ServletSecurity(@HttpConstraint(rolesAllowed = "R1"))
public class Example4 extends HttpServlet {
}
```

- Žádná omezení pro všechny metody mimo metod GET a POST, kde pro tyto metody je přístup povolen pouze rolím *R1*. U metody POST musí být navíc použit bezpečný přenos dat:

```
@ServletSecurity((httpMethodConstraints = {
@HttpMethodConstraint(value = "GET", rolesAllowed = "R1"),
@HttpMethodConstraint(value = "POST", rolesAllowed = "R1",
transportGuarantee = TransportGuarantee.CONFIDENTIAL)})
public class Example5 extends HttpServlet {
}
```

- U všech metod, mimo metodu TRACE, je přístup povolen pouze rolím *R1*. Metoda TRACE je zakázána úplně.:

```
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "R1"),
```

```
httpMethodConstraints = @HttpMethodConstraint(value="TRACE",
emptyRoleSemantic = EmptyRoleSemantic.DENY))
public class Example7 extends HttpServlet {
}
```

■

Jen pro zajímavost. Ani jedna ze tří nových anotací v přípravných verzích specifikace vůbec nefigurovala. Pro jejich účely se využívalo standardních anotací ze specifikace JSR 250, tedy anotací *DenyAll*, *PermitAll* a *RolesAllowed*. Navíc k nim měla přibýt také nová anotace *TransportProtected*. Ukázalo se však, že toto řešení má několik nedostatků, a tak se nakonec nevyužilo. Jeden z nedostatků byl například ten, že se tyto anotace váží pouze k metodám dané třídy, a tudíž nemají vliv na zděděné metody. Toto je problém, protože HTTP servlety většinou rozšiřují třídu *HttpServlet*.

3.8 Programová autentizace uživatelů

Další novinka v oblasti bezpečnosti, kterou nám nejnovější verze servletové specifikace přinesla, jsou metody *authenticate(HttpServletResponse)*, *login(String, String)* a *logout()*. Tyto metody jsou definovány v rozhraní *HttpServletRequest*, a již z jejich názvů je patrné, k jakému účelu slouží.

Metoda *authenticate* provádí autentizaci aktuálního uživatele. To, jaký typ autentizace se aktuálně využije, záleží na nastavení *login-config* elementu. Pokud je uživatel úspěšně autentizován, metoda vrátí *true* a zaručí, že metody jako *getUserPrincipal()*, *getRemoteUser* a *isUserInRole(String)* budou vracet očekávané hodnoty. Metodu *authenticate* spolu s využitím metody *isUserInRole* tedy můžeme chápat jako alternativu k elementu *auth-constraint*, což můžete vidět také na výpisech 10 a 11, pomocí nichž dosáhneme ekvivalentní funkcionality.

```
<security−constraint>
    <web−resource−collection>
        <web−resource−name>Zabezpecena sekce</web−resource−name>
        <url−pattern>URL zabezpeceneho servletu</url−pattern>
        <http−method>GET</http−method>
        <http−method>POST</http−method>
    </web−resource−collection>
    <auth−constraint>
        <role−name>admin</role−name>
    </auth−constraint>
</security−constraint>
```

Výpis 10: Nastavení autentizace pomocí *deployment descriptoru*

```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
```

```
    boolean authenticated = false;
    try {
        authenticated = request.authenticate(response);
    } catch (IllegalStateException illegalStateException) {
    } catch (IOException ioException) {
    } catch (ServletException servletException) {
    }

    if (authenticated && request.isUserInRole("admin")) {
        // byznys logika
    } else {
        response.setStatus(HttpServletResponse.SC_FORBIDDEN);
    }
}

@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    doGet(request, response);
}
```

Výpis 11: Ukázka programové autentizace

Druhou metodou je metoda *login* a je s podivem, že tato metoda byla přidána do servletů až nyní. Pomocí ní jsme totiž schopni provést kontejnerem řízenou autentizaci jmen a hesel, které jsme nějakým způsobem získali od uživatelů. Tedy od této chvíle je možné uživatele autentizovat libovolným způsobem i při využití servletového kontejneru jako správce přihlašovacích údajů. Také metody typu *getUserPrincipal*, *getRemoteUser* a *isUserInRole* budou v případě použití této metody funkční.

Metoda *login* tedy přijímá dva řetězce, a to uživatelské jméno a heslo. Kdykoliv ji zavoláte, zkusí obdržené údaje autentizovat aktuálně nastaveným přihlašovacím mechanismem. Je zřejmé, že mechanismus musí podporovat autentizaci na základě jména a hesla, neboť jsou přijímány pouze tyto dva údaje. Tedy například autentizaci klientským certifikátem nelze s metodou *login* použít. V případě, že se autentizace zdaří, metoda nastaví také platná data pro metody *getUserPrincipal*, *getRemoteUser*, *getAuthType* a *isUserInRole*. V opačném případě skončí s chybou typu *ServletException*. Ne nadarmo se říká, že metoda *login* vlastně nahrazuje přihlašování typu FORM.

Třetí metoda je metoda *logout*, která je k metodě *login* komplementární. Tedy provede odhlášení aktuálního uživatele a zaručí, že metody *getUserPrincipal*, *getRemoteUser* a *getAuthType* budou vracet hodnotu *null*.

3.9 Možnost volby techniky pro správu relací

Servlet API nově obsahuje funkcionalitu, která nám dovolí vynutit techniku pro vytváření relací dané webové aplikace. Můžeme zvolit jednu z možností *COOKIE*, *URL*, *SSL* anebo jejich kombinaci, přičemž platí, že možnost *SSL* kombinována být nemůže.

COOKIE možnost znamená, že se použijí standardní cookie, *URL* vynucuje techniku přepisování adres a *SSL* nařizuje využití SSL protokolu, ze kterého jsou následně získána data, která jednotlivé uživatele jednoznačně identifikují.

API je reprezentováno metodami *setSessionTrackingModes(Set<SessionTrackingModes>)*, *getEffectiveSessionTrackingModes()*, *getDefaultSessionTrackingModes()* v rozhraní *ServletContext* a výčtem *javax.servlet.SessionTrackingMode*. První metodou nastavujeme techniky, které požadujeme, aby servletový kontejner aktuálně využíval, druhou můžeme zjistit, jaké techniky jsou v současné době používány a třetí nám vrátí techniky, které kontejner používá implicitně. Metodu *setSessionTrackingModes* lze však volat jen do té doby, než je daná webová aplikace inicializována.

3.10 Podpora konfigurace atributů kontejnerem generovaných cookie

Od této chvíle již žádné *JSESSIONID*! Samozřejmě tato věta je myšlena pouze jako nadsázka toho, že, od nejnovější verze servletové specifikace, jsme schopni nastavit také atributy těch cookie, které nevytváříme sami, ale generuje je samotný servletový kontejner pro správu relací, tedy v případě HTTP to jsou právě ony *JSESSIONID*. A proč již žádné *JSESSIONID*? To z toho důvodu, že je dovoleno měnit i název těchto generovaných cookie na libovolnou hodnotu, která ale, samozřejmě, musí být v souladu se specifikací RFC 2109, jenž HTTP cookie definuje.

Konfigurace jednotlivých atributů je prováděna přes nové rozhraní *javax.servlet.SessionCookieConfig*, které je vytvářeno v jedné instanci pro každou webovou aplikaci. Toto rozhraní lze získat metodou *getSessionCookieConfig()* na objektech typu *ServletContext*. Jediné omezení je v tom, že nastavovat jednotlivé atributy nemůže kdykoliv, ale pouze do té doby, než je daná webová aplikace inicializována, respektive inicializován odpovídající *ServletContext* objekt. Tedy nastavení jednotlivých atributů obvykle provedeme v metodách *contextInitialized(ServletContextEvent) ServletContextListener* posluchačů.

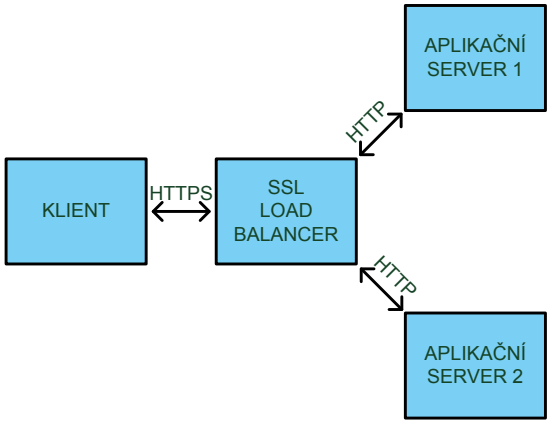
Samozřejmě je jasné, že naprostá většina z nás tuto novou funkcionalitu zřejmě nevyužije[20], ovšem v určitých případech se může hodit. Představte si například situaci, kdy máte několik aplikačních serverů, před které je postaven SSL load balancer pro řízení zátěže. Jednotlivé aplikační servery s load balancerem komunikují pomocí čistého HTTP a on samotný komunikuje s klienty pomocí HTTPS (tuto topologii můžete vidět na obrázku 4[21]). Zde je nutné, aby i cookie, které vytvoří jednotlivé aplikační servery, byly označeny jako *secure* a byly tak chráněny i na cestě mezi klientem a load balancerem.

3.11 Podpora HttpOnly cookies

Další změnou týkající se cookie dat a bezpečnosti, je podpora nového *HttpOnly* atributu, který lze nastavit jak pro vlastní cookie, tak pro cookie generované servletovým kontejnerem.

[20] Například samotná specifikace říká, že měnit název těchto cookie bychom měli jen ve výjimečných případech, neboť to může mít za následek nekompatibilitu vyšších vrstev, které výchozí hodnotu, tj. *JSESSIONID*, očekávají.

[21] Obrázek byl převzat z této adresy: http://blogs.sun.com/jluehe.



Obrázek 4: Topologie serverů příhodná k programové změně parametrů cookie

Pokud tento atribut uvedete[22], znamená to, že daná cookie nebude dostupná klientským skriptům a její odcizení tak bude o něco složitější. Bohužel tento atribut doposud není součástí žádné specifikace, a tak není implementován všemi prohlížeči. Další nevýhodou je fakt, že *HttpOnly* cookie nejsou nijak chráněny v případě využití *XMLHTTPRequest* objektu, ale snad se i tento nedostatek podaří vyřešit[23].

I přes výše zmíněné nedostatky, je použití *HttpOnly* atributu pro relační cookie výhodné již nyní. Ty prohlížeče, které jej neznají, ho ignorují a navíc *XMLHTTPRequest* taktéž není problémem, nebot'relační cookie se obvykle zasílají pouze jednou, a tudíž pro následné *XMLHTTPRequest* požadavky již zaslány nebudou. Těchto faktů si byli samozřejmě vědomi také lidé z expertní skupiny, a tak servletová specifikace obsahuje větu, která každému 3.0 kontejneru nařizuje poskytovat možnost nastavení výchozí hodnoty *HttpOnly* atributu. Tato hodnota se pak využije pro všechny cookie, u kterých nebyl *HttpOnly* atribut explicitně stanoven.

Podrobnější popis *HttpOnly* atributu a tabulku prohlížečů, které jej implementují, můžete najít na stránkách OWASP[24].

3.12 Podpora HTTP požadavků typu multipart/form-data

Pokud jste někdy programovali servlet, který zpracovával soubory zaslané HTML formuláři[25], jistě víte, že tento proces nebyl až tak přímočarý, jak by mohl být. Do minulé verze servletů existovaly v podstatě dvě možnosti, jak HTTP POST požadavky typu *multipart/form-data*, kterých se při zasílání HTML formulářů se soubory využívá, zpracovat. První, a obvyklou, metodou bylo využití některé z knihoven, které tuto funkcionalitu

[22] Uvádí se bez jakékoliv hodnoty, tak jako například atribut *secure*.

[23] Viz například https://lists.owasp.org/pipermail/webappsec/2008-May/000597.html.

[24] Viz URL http://www.owasp.org/index.php/HTTPOnly.

[25] Specifikace k uploadu souborů lze najít zde: http://www.w3.org/TR/html401/interact/forms.html#h-17.13.4.2 a zde: http://www.ietf.org/rfc/rfc2388.txt.

již nabízí[26] a druhou, zpracování požadavku dle RFC 2388 vlastními silami, což ale není až tak jednoduché, jak by se mohlo na první pohled zdát (stačí si projít zdrojové kódy výše zmíněných knihoven). Zřejmě nejpoužívanější knihovna byla ta od Jasona Huntera[27], který se v prostředí servletů pohybuje již dlouho a je členem expertních skupin, které jsou zodpovědné za vytvořením jednotlivých servletových specifikací.

Od této verze servletové specifikace žádnou externí knihovnu pro zpracování *multipart/form-data* požadavků již používat nemusíme, jelikož bylo přidáno nové API, které tuto funkcionalitu realizuje. S největší pravděpodobností za tímto API stojí právě Jason Hunter a projekt Commons Upload, jelikož jeho podoba se od původní knihovny Jasona a Commons Upload API příliš neliší.

API je velmi jednoduché a je reprezentováno metodami *getParts()* a *getPart(String)* v rozhraní *HttpServletRequest* a anotací *MultipartConfig* z nového balíčku *javax.servlet.annotation*. Princip je takový, že pokud je určitý servlet anotován *MultipartConfig* anotací a příchozí požadavek je typu *multipart/form-data*, může programátor v daném servletu (popřípadě ve filtru, který je pro daný servlet přiřazen) volat výše zmíněné metody pro získání objektů typu *javax.servlet.http.Part*. Tyto objekty reprezentují jednotlivé prvky odeslaného formuláře. Na výpisu 12, respektive 13 můžete vidět ukázkový HTML formulář, který odesílá jméno a obsah jednoho souboru, respektive kód servletu, který ho zpracovává.

```
<form action="cesta/k/FileUploadServlet" method="post"
enctype="multipart/form-data">
    <p>Nazev souboru: <input name="nazev" type="text"/></p>
    <p>Soubor: <input name="soubor" type="file"/></p>
    <p><input type="submit" value="Odeslat soubor"/></p>
</form>
```

Výpis 12: HTML formulář pro upload souborů

```
@MultipartConfig(location = "C:\\uploads", maxRequestSize = 2000000,
maxFileSize = 1000000, fileSizeThreshold = 50)
public class FileUploadServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        try {
            Part nazev = request.getPart("nazev");
            Part soubor = request.getPart("soubor");

            InputStream nazevSouboru = nazev.getInputStream();
            InputStream obsahSouboru = soubor.getInputStream();

            // kod, ktery zpracovava obsah jednotlivych casti
        } catch (ServletException servEx) {
```

[26] Soupis těchto knihoven je dostupný například zde: http://www.jguru.com/faq/view.jsp?EID=160.
[27] Knihovna je stále dostupná ke stažení pod URL http://www.servlets.com/cos/.

```
            // pozadavek neni typu multipart/form-data
        } catch (IOException ioEx) {
            // nastala I/O chyba pri zpracovani pozadavku
        } catch (IllegalStateException illStEx ) {
            // telo pozadavku je vetsi nez 2MB nebo nektera z casti je vetsi
            nez 1MB (tyto velikosti jsou nastavene v anotaci MultipartConfig)
        }
    }
}
```

Výpis 13: Použití *MultipartConfig* anotace

Rozhraní *Part* je velmi prosté a nabízí několik metod, pomocí nichž můžeme získat různé informace o dané části (např. její obsah, jméno, velikost, datový typ) anebo danou část zapsat na pevný disk.

Pomocí anotace *MultipartConfig* jsme zase schopni definovat, do jaké složky se budou ukládat dočasné soubory pro jednotlivé části (nebo soubory vytvořené metodou *write(String)* v rozhraní *Part*), velikostní limity jak pro celý požadavek tak části a v neposlední řadě také velikost, od které se pro danou část vytvoří soubor, a tudíž nebude servletovým kontejnerem uchovávána v paměti.

4 Realizace vlastního servletového kontejneru

Hlavním úkolem této práce, vedle popisu novinek v nejnovější verzi servletové specifikace, byla realizace vlastního jednoduchého servletového kontejneru. Takovýto kontejner jsem tedy implementoval (pojmenoval jsem jej, možná trošku troufale, „*Pike*“, což česky znamená „štika“) a proces jeho realizace je uveden v následujících třech podkapitolách. Předem bych chtěl ale zdůraznit, že vytvořený kontejner je pouze takový „Proof-of-Concept“, jak by se řeklo v terminologii technologie RUP[28], nebot' si neklade za cíl být se servletovou specifikací plně kompatibilní a tím pádem vyhovět TCK[29] pro JSR 315, ale pouze podat náhled, jakým způsobem mohou být stěžejní části technologie Java Servlet 3.0 implementovány.

Poněvadž servletových kontejnerů, které implementují minulé verze servletů, již existuje několik, a většinou také mívají veřejně dostupné zdrojové kódy, realizoval jsem svůj kontejner tak, aby nabízel pouze základní funkcionalitu z předešlých verzí servletové specifikace (jakou přesně, je uvedeno v následujících podkapitolách), ale, oproti tomu, umožnil využití hlavních novinek z verze 3.0.

Vlastní realizaci jsem tedy rozdělil celkem do tří kroků. V tom prvém jsem nejprve položil základ celého kontejneru, na němž jsem následně ve druhém a třetím kroku stavěl. Tedy výstupem první části byla základní architektura, která umožňovala pouze primitivní obsluhu HTTP požadavků vlastními silami a o servletech doposud nic nevěděla. Servlety (tj. servletové API a sémantiku servletové specifikace) jsem realizoval až v kroku druhém, který již, na svém konci, obsahoval plnohodnotný servletový kontejner, který umožnil práci se základními prvky Servlet 2.5 technologie. Ve třetím, a posledním kroku, jsem postupně realizoval jednu servletovou novinku za druhou, kde pořadí realizace jsem stanovil tak, abych vždy, při vývoji následující novinky, mohl navázat na již vytvořené API. Tímto jsem tedy aplikoval jednu ze šesti doporučených praktik při tvorbě software, a to iterativní způsob vývoje.

Realizace prvního kroku je uvedena v podkapitole 4.1, druhý krok je popsán v podkapitole 4.2 a konečně třetí v podkapitole 4.3. Tyto podkapitoly jsou dále členěny do menších částí, přičemž platí, že každá část do kontejneru většinou přidává novou funkcionalitu a vytváří tak jeho další verzi. Tedy tyto části lze zjednodušeně chápat jako realizace jednotlivých iteračních cyklů. Pro snazší studium jednotlivých změn, které každá iterace přinesla, jsem vždy, po jejím konci, vytvořil kopii aktuálních zdrojových kódů a tyto data následně umístil na přiložené CD[30]. Dále je nutné říci, že v textu neuvádím žádné implementační detaily, kterých jsem se při vývoji kontejneru *Pike* musel držet, nebot'si myslím, že by tyto informace byly nadbytečné a zabraly by navíc příliš mnoho stránek. To, co v textu popisuji, je architektura kontejneru z „ptačí“ perspektivy. K tomuto účelu je, krom

[28]Rational Unified Process je softwarový proces původně vyvinutý společností Rational Software Corporation, která je dnes divizí firmy IBM, jehož hlavní devízou je iterativní způsob vývoje software. Pro více informací o tomto procesu, navštivte, prosím, tuto URL: http://www.ibm.com/software/awdtools/rup/.

[29]Technology Compatibility Kit je sada testů, která se vytváří pro každé JSR, a jenž má za úkol prověřit, zda jsou její implementace ve shodě se specifikací. Více informací viz http://jcp.org/en/procedures/jcp2.

[30]Zdrojové kódy jsou dostupné ve formě projektů do Eclipse IDE, kde byla využita verze 3.5.1. Eclipse IDE si můžete stáhnout z této adresy: http://www.eclipse.org/downloads/.

textového popisu, také mnohdy využito služeb UML[31], což je nástroj, k vytváření abstraktních modelů, velmi vhodný. Při návrhu architektury jsem se také snažil praktikovat komponentní přístup, který je, spolu s vizuálním modelováním, další z doporučovaných technik pro vývoj software.

4.1 Návrh základní architektury

Jak jsem již uvedl, tato podkapitola popisuje návrh a realizaci základní architektury vlastního servletového kontejneru. Abychom však byli schopni tuto architekturu vůbec vytvořit, musíme nejprve identifikovat její jednotlivé části. Tedy nelze začít ničím jiným, než stanovením hlavních funkčních a nefunkčních požadavků, které bude muset vytvářený kontejner, respektive základní architektura realizovat[32]. Je jasné, že pro stanovení požadavků nelze použít klasických technik jako například rozhovorů s budoucími uživateli systému, jelikož je realizována specifikace a nikoliv aplikační software. Tudíž požadavky musíme získat odjinud a jinými způsoby. Jelikož však implementujeme specifikaci, pro kterou navíc již existují funkční realizace (i když třeba pro starší verze), je zřejmé, odkud požadavky získáme. Získáme je celkem ze tří zdrojů, a to ze servletové specifikace, servletového API a již existujících servletových kontejnerů, u kterých lze s výhodou použít empirie.

Pokud si tyto zdroje letmo prostudujete, jistě Vás ihned napadnou stěžejní funkční a nefunkční požadavky na servletové kontejnery, které jsou uvedeny v tabulce 1[33]. Všechny tyto požadavky, mimo požadavků *REQ0003* a *REQ0006*, které se týkají pouze servletů, a jimiž se tedy zabývám až ve druhém kroku, musí zcela jistě základní architektura každého servletového kontejneru realizovat. Tyto požadavky tedy dohromady tvoří všechny její části. Skládá se tudíž z webového serveru, který poskytuje komunikační kanály, přes které jsou přijímány HTTP požadavky a odesílány HTTP odpovědi, souborů a tříd, které dokážou servletový kontejner spustit a ukončit, dále ze základního konfiguračního subsystému a v neposlední řadě také ze samotné distribuce. Obsahem této podkapitoly je tedy návrh a realizace těchto jednotlivých požadavků.

4.1.1 HTTP server

Tato podkapitola je první iterací a již z jejího názvu můžete odhadnout, že se zabývá realizací HTTP serveru, tj. požadavku *REQ0004*. Webovým serverem začínám z toho důvodu, protože tvoří „páteř“ každého servletového kontejneru, na kterou se pak snadno

[31] Unified Modeling Language je modelovací jazyk, pomocí nějž jsme schopni dokumentovat vyvíjený systém. Více informací, o této technologii, lze najít například zde: http://www.uml.org/.

[32] Při návrhu běžných aplikací bychom zřejmě ihned nezačali stanovením požadavků, ale identifikací jednotlivých aktérů budoucího systému. Nicméně v případě servletového kontejneru je to zbytečné, neboť funkčních požadavků, a tím pádem také aktérů, je velmi málo. Klasickými aktéry zde mohou být například „Koncový uživatel“ a „Operátor“, kde koncovým uživatelem je myšleno něco, co na kontejner zasílá HTTP požadavky a přijímá HTTP odpovědi a operátorem pak někdo, kdo má fyzický přístup k samotné distribuci kontejneru a je schopen jej například zastavit či konfigurovat.

[33] U požadavků je uveden pouze jejich název a krátký popis, jelikož si myslím, že jsou tak zřejmé, aby tyto informace plně dostačovaly. Neuvádím tedy například jejich scénáře, primární aktéry atd.

ID	Typ	Název	Popis	Původ
REQ0001	Funkční	Spuštění a ukončení servletového kontejneru	Servletový kontejner musí jít spustit a také ukončit.	Existující servletové kontejnery
REQ0002	Funkční	Konfigurace servletového kontejneru	Servletový kontejner musí jít konfigurovat.	Existující servletové kontejnery, servletová specifikace
REQ0003	Funkční	Nahrání a odebrání webové aplikace	Servletový kontejner musí umožnit nahrání a případné odebrání webových aplikací.	Servletová specifikace
REQ0004	Funkční	Zpracování HTTP požadavků	Servletový kontejner musí být schopen obsloužit HTTP požadavky směřující na jednotlivé webové aplikace.	Servletová specifikace
REQ0005	Nefunkční	Servletová specifikace	Servletový kontejner musí být realizován v souladu se specifikací JSR 315.	Servletová specifikace
REQ0006	Nefunkční	Servletové API	Servletový kontejner musí implementovat servletové API.	Servletová specifikace
REQ0007	Nefunkční	Distribuce servletového kontejneru	Servletový kontejner musí být distribuován jako adresář různých souborů a složek, které umožní správnou funkčnost nahraných webových aplikací.	Existující servletové kontejnery

Tabulka 1: Základní požadavky na navrhovaný servletový kontejner

přidávají další funkcionality. Tím je tedy elegantně (a velmi jednoduše) zaručen iterativní postup při vývoji.

Avšak specifikace říká, že servletový kontejner je vlastně pouze část webového serveru, která obsluhuje uživatelské požadavky a spravuje životní cyklus servletů. Tedy k tomu, abychom úspěšně realizovali servletový kontejner, nutně nemusíme realizovat také samotný webový server. Jinými slovy, servletový kontejner je, na webovém serveru, poměrně nezávislá komponenta, která jej využívá pouze k naplnění svého účelu. Samozřejmě ale platí, že servletový kontejner by bez webového serveru postrádal smyslu, jelikož by nebyl schopen obsloužit ani jeden uživatelský požadavek.

Z výše uvedených důvodů, HTTP server neimplementuji sám, ale využiji již existujícího řešení. Navíc implementovat výkonný HTTP server není vůbec nic jednoduchého a také jeho návrh by byl poměrně rozsáhlý. Ovšem řešení, ze kterých mohu vybírat, není až tak mnoho, neboť jsem nucen dodržet také ostatní požadavky, přesněji požadavky *REQ0005* a *REQ0007*. Požadavek *REQ0007* nařizuje, aby byl kontejner *Pike* dostupný ve formě kompaktní distribuce, která se prostě jen nakopíruje na daný počítač, následně se spustí a ihned bude schopna obsluhovat uživatelské požadavky. Nebude se tedy muset nijak konfigurovat a ani, ke své činnosti, nebude využívat nějaký externí software. Z těchto důvodů je nejvhodnější, aby byl HTTP server plně zabudovatelný, tedy realizován jako prosté API. Toto API navíc musí být naprogramováno v Javě, neboť tento požadavek zase nařizuje servletová specifikace, kterou jsem nucen (na základě požadavku *REQ0005*) dodržet. Ta navíc požaduje, aby server také podporoval HTTP protokol ve verzi 1.1.

Jako možné řešení HTTP serveru mě tedy napadají tyto způsoby:

- Servery z již existujících kontejnerů. První možností je prosté vyextrahování HTTP serveru z jednoho ze současných servletových kontejnerů (samozřejmě pouze za předpokladu, že to dovolí jeho licence). HTTP server můžeme vyextrahovat například z kontejneru Jetty[34] či Tomcat[35].

- Miniaturní HTTP servery. Další možnost je využití jednoho z mnoha miniaturních Java HTTP serverů. Lze použít například balíček *com.sun.net.httpserver*, který je integrovaný přímo do Sun JavaSE.

- Nadstavby NIO frameworků. Do této skupiny můžeme zařadit například projekty jako AsyncWeb[36] či Grizzly, což jsou vysoce výkonné HTTP servery.

Jako nejlepší řešení se mi zdá právě projekt Grizzly[37], využiji tedy jej. Grizzly je NIO framework, což je platforma určená k vytváření vysoce výkonných a snadno škálovatelných serverů. Škálovatelnost je dosažena především díky tomu, že je, stejně jako u ostatních NIO frameworků, využito asynchronního vstupu a výstupu (prostřednictvím

[34]Jetty je novější, ale dnes již velmi oblíbený servletový kontejner. Jeho domovská stránka je přístupná pod touto adresou: http://eclipse.org/jetty/.

[35]Tomcat je zcela jistě nejznámější a, i když se to tak, na první pohled, nemusí jevit, doposud také nejpoužívanější servletový kontejner, jelikož jej interně využívá většina aplikačních serverů. Tento kontejner je dostupný pod URL: http://tomcat.apache.org/.

[36]Více informací viz http://mina.apache.org/asyncweb/.

[37]Projekt Grizzly je dostupný pod touto URL: https://grizzly.dev.java.net/.

balíčků *java.nio*) a realizován návrhový vzor Reactor[38]. Díky tomuto faktu, jsou NIO frameworky dokonalou ukázkou realizace principu vlákna pro každý požadavek (viz podkapitola 3.1.2). Avšak hlavním důvodem, proč volím právě projekt Grizzly, není až tak jeho výkon, ale fakt, že je zároveň i HTTP serverem, jelikož nabízí také HTTP nadstavbu.

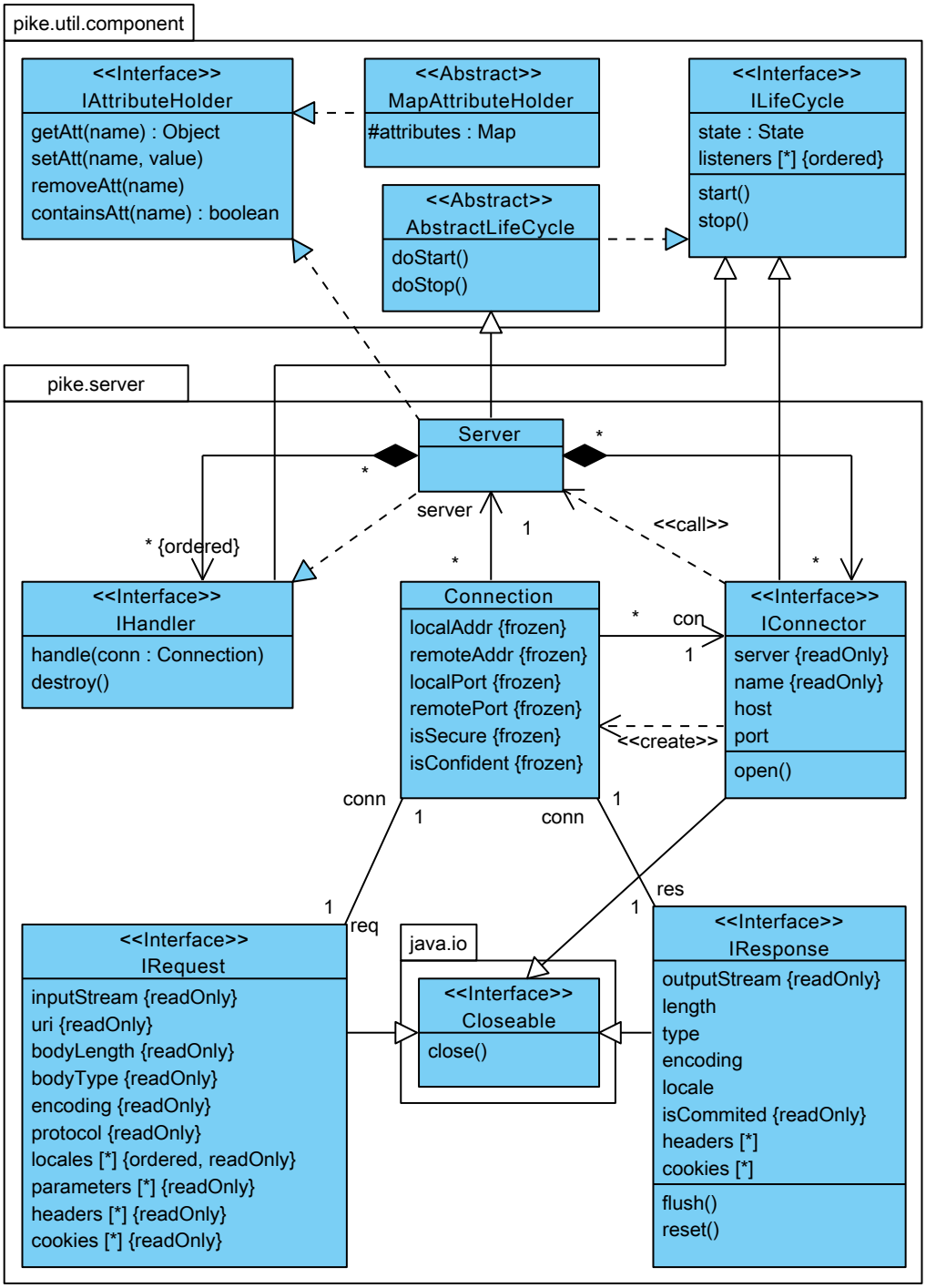
4.1.1.1 Klíčové prvky základní architektury

HTTP server máme již zvolený, můžeme se tedy pustit do návrhu klíčových rozhraní a tříd architektury kontejneru *Pike*, která bude jeho reprezentací. Jelikož však chci vyvíjet komponentně, základní architekturu navrhnu obecně. Ta tedy nebude přímo vázána na Grizzly server, ale pouze nadefinuje sadu tříd, které se následně budou muset implementovat tak, aby interně využívaly právě projekt Grizzly.

Není nutné, aby architektura obsahovala mnoho tříd. Stačí, když nabídne abstrakci obecného serveru a komunikačních kanálů, které se poté budou moci realizovat pomocí frameworku Grizzly. Architektura se tedy bude skládat z prvků, které můžete vidět na obrázku 5[39]. Centrálním bodem je jistě třída *Server*, jejíž zodpovědnost, ale paradoxně není příliš veliká (což je elegantně zapříčiněno OOP principy). Jejím primárním účelem je agregovat konektory a obsluhovače, což jsou další dva stěžejní prvky architektury.Konektor, jež je reprezentován rozhraním *IConnector*, je něco, co umí přijímat klientské požadavky a následně je předávat serveru. To, jakým způsobem a jaké typy požadavků příjme, není pro architekturu důležité a je jí skryto. Každý konektor pro každý nový požadavek je totiž nucen vytvořit objekt třídy *Connection*. Tento objekt je unikátní pro každý požadavek a krom odkazu na konektor, který jej vytvořil, také obsahuje referenci na server daného konektoru, objekt typu *IRequest* a objekt typu *IResponse*. Již z názvů těchto dvou tříd vyplývá, co reprezentují. Rozhraní *IRequest* je abstrakcí požadavku, který konektor přijal, a který bude kontejnerem *Pike* obsluhován. Toto rozhraní tedy definuje minimální kontrakt pro všechny typy požadavků, které budou konektory přijímat. Rozhraní *IResponse* definuje totéž pro odpovědi. I když to z diagramu není zřejmé, tyto dvě rozhraní budou velmi podobné typům *HttpServletRequest* a *HttpServletResponse*, poněvadž není potřeba vymýšlet nějakou přehnanou abstrakci a většina konektorů bude stejně operovat pouze nad HTTP protokolem. Pokud vím, tak tento princip razí také mnoho ostatních kontejnerů, i když základ servletů je navržen zcela obecně. Avšak je nutné si říci, že, mimo HTTP, se servlety příliš neujaly, alespoň co se týče mých dosavadních znalostí a zkušeností. Druhým stěžejním prvkem jsou obsluhovače (objekty implementující rozhraní *IHandler*). Stejně jako je to u konektorů, i těchto objektů může server obsahovat více a jsou zodpovědné za obsluhu požadavků, respektive objektů typu *Connection*. Tzn. kdykoliv je konektorem přijat nový požadavek, vytvoří se nový objekt *Connection*, předá se serveru a ten jej následně (sekvenčně) postoupí všem registrovaným obsluhovačům. Ty poté mohou využít objekt *IResponse* pro zápis odpovědi. Dalším neméně důležitým prvkem architektury je rozhraní

[38] Více informací viz http://en.wikipedia.org/wiki/Reactor_pattern.

[39] Obrázek 5 je třídní diagram s vyšší mírou abstrakce, který má za cíl podat pouze hrubý náhled na hlavní třídy a rozhraní. Tedy neuvádí přesné signatury atributů a metod a ani je neuvádí všechny. Omezení {frozen} značí konstanty, omezení {readOnly} znamená, že třída nebude poskytovat žádné API pro změnu či nastavení daného atributu. Tento typ třídního diagramu budu využíván i nadále.



Obrázek 5: Klíčové třídy základní architektury kontejneru *Pike*

ILifeCycle, které, pokud jej libovolná třída realizuje, značí, že je schopna startu a zastavení. Toto rozhraní tedy reprezentuje životní cyklus komponent a implementují jej jak *Server*, tak obsluhovači a „konektoři".

4.1.1.2 Návrh Grizzly konektoru

Rozhraní pro konektory je již definováno, nyní je potřeba jej také implementovat. Jak jsem již uvedl, implementaci provedeme pomocí frameworku Grizzly, který jsem zvolil pro realizaci HTTP služeb v kontejneru *Pike* (samozřejmě kdokoliv jiný může přidat také další realizace). Implementace bude velmi jednoduchá, neboť nám stačí realizovat pouhé tři rozhraní, a to *IConnector*, *IRequest* a *IResponse*.

Začněme rozhraními *IRequest* a *IResponse*. Jelikož framework Grizzly již využívá své třídy pro požadavky a odpovědi (pro požadavky definuje třídu *GrizzlyRequest*, pro odpovědi *GrizzlyResponse*), které nám nabízejí také totožnou funkcionalitu, rozhraní *IRequest* a *IResponse* nerealizuji sám, ale využiji návrhového vzoru Adapter[40]. Tedy vytvořím třídy *RequestAdapter*, respektive *ResponseAdapter*, které budou implementovat rozhraní *IRequest*, respektive *IResponse* tak, že v konstruktoru přijmou objekty typu *GrizzlyRequest*, respektive *GrizzlyResponse* a ve všech svých metodách budou pouze volat odpovídající metody na těchto typech. Jejich realizace bude tedy primitivní, ale elegantní.

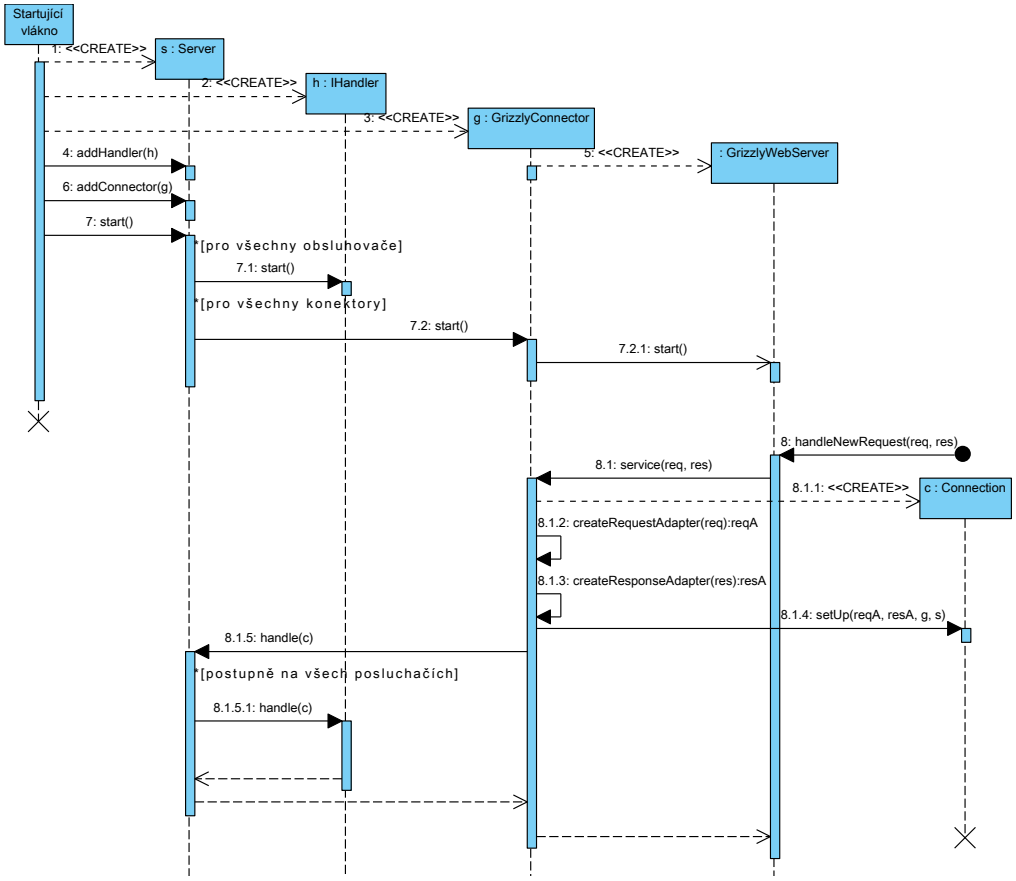
A nejinak tomu bude i v případě implementace rozhraní *IConnector*. Ta pouze využije další Grizzly třídu, a to *GrizzlyWebServer*. Tu v konstruktoru patřičně nakonfiguruje a v metodě *doStart*, kterou definuje třída *AbstractLifeCycle*, a jež se bude volat při startu konektoru (popisuji dále), zavolá její metodu *start*. Metoda *start* zajistí, že se začne naslouchat na HTTP požadavky. Pro více informací se, prosím, podívejte do dokumentace projektu Grizzly (konkrétně na třídu *com.sun.grizzly.http.embed.GrizzlyWebServer*) a na implementaci třídy *pike.server.conn.grizzly.GrizzlyConnector*, která je dostupná jak ve zdrojových kódech první iterace, tak zobrazena v příloze C (konkrétně v podkapitole C.1).

4.1.1.3 Tok akcí mezi prvky architektury

Jak říká Martin Fowler, ve své známé knize o UML[5], zaměřit se pouze na strukturu bez analýzy chování je nesmysl, proto zde popíši, jak spolu (a v jakém pořadí) jednotlivé prvky architektury komunikují.

Chování architektury za běhu je krásně vidět na obrázku 6. K tomu, abychom architekturu vůbec spustili (tj. kontejner *Pike*), musíme nejprve vytvořit instanci objektu *Server*. To lze velmi snadno, neboť je to neabstraktní veřejná třída. Poté je nutné serveru předat alespoň jeden konektor a obsluhovač. Konektor vytvoříme instancionováním třídy *GrizzlyConnector* (viz předešlý odstavec), kterému předáme port k naslouchání, a obsluhovač realizací rozhraní *IHandler*, což je proces téměř totožný s realizací servletu. Jakmile toto provedeme, stačí na serveru zavolat metodu *start*, která se již postará o vše ostatní. Tedy o nastartování všech obsluhovačů (voláním jejich metod *start*) následované nastartováním všech konektorů (taktéž voláním jejich metod *start*). Co třída *GrizzlyConnector* ve své metodě *start* provede, již víme, nastartuje nakonfigurovaný *GrizzlyWebServer*, čímž se na daném portu začne naslouchat na HTTP požadavky. Pokud bychom serveru žádný

[40] Více informací o tomto vzoru můžete nalézt zde: http://en.wikipedia.org/wiki/Adapter_pattern.



Obrázek 6: Tok akcí mezi prvky základní architektury kontejneru *Pike*

konektor nepřidělili, nic by se nestalo, pouze by daný Java proces ihned skončil (*GrizzlyWebServer* totiž naslouchá na svých vláknech), samozřejmě pouze v případě, že by již dříve nespustil nějaká jiná vlákna.

Obsluha požadavků probíhá takto. Kdykoliv libovolný konektor přijme nový požadavek, vytvoří pro něj objekt typu *Connection* a zavolá na svém přiděleném serveru metodu *handle(Connection)*. Tato metoda následně volá své jmenovkyně na všech obsluhovačích, které byly danému serveru předány. Ty si následně mohou z přijatého objektu typu *Connection* získat implementace rozhraní *IRequest*, respektive *IResponse* (v případě konektoru Grizzly to budou, výše zmíněné, adaptéry) a provést požadované akce. Tímto je proces zpracování jednoho požadavku ukončen. Kdykoliv je přijat požadavek nový, provede se opět to samé.

4.1.2 Distribuce kontejneru Pike

Další požadavek, který je nutné splnit, je požadavek *REQ0007*. Ten nařizuje vytvoření distribuce pro servletový kontejner *Pike*. Distribuce není nic jiného než prostá složka

obsahující dokumentaci, dávky pro spuštění a ukončení kontejneru, knihovny, složku pro webové aplikace, soubory s nastaveními atd.

Pokud se podíváte na distribuce známých kontejnerů jako jsou Jetty či Tomcat, zjistíte, že obsahují zhruba tyto adresáře:

- složka *bin*. Tato složka většinou obsahuje artefakty nutné ke spuštění (a popřípadě také ukončení) kontejneru.
- složka *conf*. Obsahuje různé soubory s konfigurací.
- složka *docs*. Obsahuje uživatelskou (a někdy také programátorskou) dokumentaci.
- složka *lib*. Obsahuje programový kód, který je nutný pro běh samotného kontejneru. Ten je většinou dále rozdělen do mnoha JAR souborů, jelikož kontejnery obvykle využívají také mnoho knihoven třetích stran.
- složka *logs*. Slouží k uchování souborů se zalogovanými zprávami. Zprávy loguje jak kontejner samotný, tak jednotlivé webové aplikace (prostřednictvím rozhraní *ServletContext*).
- složka *temp*. Uchovává případné dočasné soubory.
- složka *webapps*. Uchovává samotné webové aplikace.
- složka *work*. Pracovní složka, do které si kontejner ukládá libovolná data, která si během svého běhu potřebuje perzistovat.

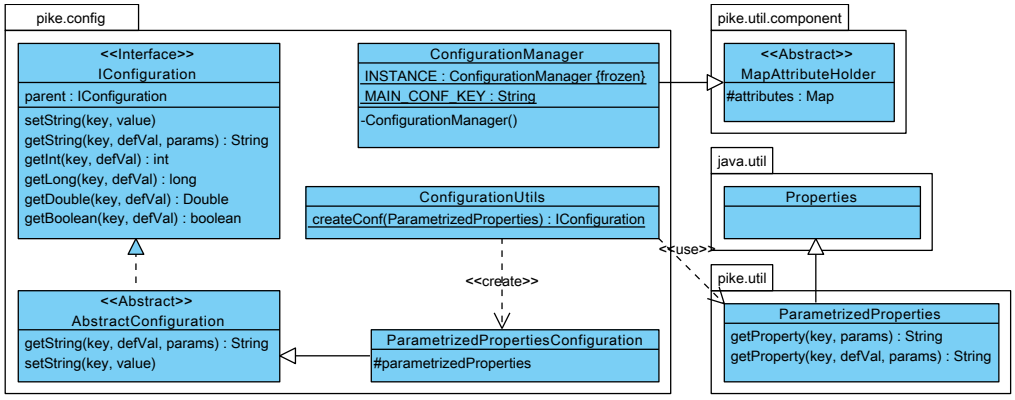
Ani kontejner *Pike* nebude výjimkou a bude všechny tyto adresáře také obsahovat. Pro přesnou podobu jeho distribuce, se, prosím, podívejte na druhou iteraci, která je dostupná na přiloženém CD (tato iterace vznikla právě realizací požadavku *REQ0007*). Samozřejmě, že v tuto chvíli distribuce ještě téměř nic neobsahuje, postupně však bude plněna daty, která budou potřeba pro implementaci následujících iterací.

4.1.3 Konfigurační subsystém

Požadavek *REQ0002* nařizuje, aby bylo možné kontejner *Pike* konfigurovat. Toto je logické, neboť bez konfigurace se zcela jistě neobejde žádný ze servletových kontejnerů.

Konfigurace bude založena na jednoduchých Java *properties* souborech, avšak kontejner bude obsahovat pouze jeden konfigurační soubor, jelikož více jich prozatím není potřeba. Tento soubor bude pojmenován *pike.properties* a bude umístěn ve složce *conf*, která byla vytvořena v předešlé iteraci. Soubor bude využíván všemi částmi kontejneru, které bude možné nějak konfigurovat. Těmito částmi jsou například server, konektory a obsluhovače.

Aby však bylo ke konfiguraci přistupováno jednotně, je třeba, aby se navrhlo jednoduché konfigurační API, které jako jediné bude mít přístup k *pike.properties* souboru, a jež všem ostatním částem zprostředkuje přístup ke konfiguračním rutinám. Takovéto API můžete vidět na obrázku 7.



Obrázek 7: Konfigurační API kontejneru *Pike*

API je opět navrženo obecně pomocí rozhraní. Každá konfigurace (tedy i *pike.properties* soubor) bude za běhu reprezentována jedním objektem typu *IConfiguration*. Tento typ specifikuje několik metod, pomocí nichž je možné do konfigurace uložit nebo načíst řetězec. Dále také nabízí konvenční metody pro načtení čísel a booleovské hodnoty. Všimněte si, že také obsahuje možnost načtení řetězce parametrizovaně (tj. s použitím vstupních parametrů). Toto je velmi žádaná vlastnost například při formátování různých výstupních zpráv. Dále je v API obsažena konvenční abstraktní třída *AbstractConfiguration*, která realizuje téměř všechny metody rozhraní *IConfiguration* a jako abstraktní ponechává pouze metody pro nastavení, respektive načtení řetězce. Jelikož *properties* soubory nepodporují parametrizované data, API také obsahuje třídu *ParametrizedProperties*, která tento nedostatek řeší. Ovšem díky elegantnějšímu návrhu, tato třída přímo *AbstractConfiguration* nerozšiřuje, ale tento úkol přenechává typu *ParametrizedPropertiesConfiguration*. Ten již konfigurací je, kterou realizuje právě za pomoci *ParametrizedProperties*. Je tedy ukázkovým příkladem použití návrhového vzoru adaptér. Aby však tvůrci konfigurací nebyli na tuto třídu vázání přímo, API obsahuje továrnu s názvem *ConfigurationUtils*. Ta deklaruje statickou metodu, která přijímá parametrizované *properties*, pro něž vrací implementaci *IConfiguration*. Posledním prvkem konfiguračního API je singleton *ConfigurationManager* řešící onen jednotný přístup. Jeho princip je takový, že jelikož rozšiřuje, již zmíněnou, třídu *MapAttributeHolder* (viz obrázek 5), bude sloužit k uchovávání všech *IConfiguration* objektů. Tedy v případě, že určitý kód bude chtít získat nějakou konfiguraci, vždy na tomto singletonu zavolá metodu *getAtt* s patřičným klíčem. Jelikož máme pouze jednu hlavní konfiguraci, třída dále obsahuje statický člen, který obsahuje klíč, pod kterým bude za běhu schovaná konfigurace načtená z *pike.properties* souboru. Tato konfigurace se vytvoří a nastaví při startu kontejneru.

4.1.4 API pro start a ukončení kontejneru

Poslední část základní architektury, jež se netýká přímo servletů, a kterou je tedy nutné nyní realizovat, je subsystém zodpovědný za start a následné ukončení kontejneru. Tvorbu tohoto subsystému nám ostatně také nařizuje požadavek *REQ0001*.

Pokud se začtete do programátorských dokumentací moderních servletových kontejnerů, zjistíte, že je jejich start, oproti startu běžné Java aplikace, mírně složitější. To je zapříčiněno tím, že servletové kontejnery si, pro nahrávání svých tříd, vytváří své vlastní *Classloadery*. K tomuto účely tedy nevyužívají proměnnou *classpath*, která se používá ke specifikaci všech tříd, které mají být dostupné danému JVM při jeho startu. Proměnná *classpath* je kontejnery většinou nastavena na jediný JAR soubor (často je nazýván *bootstrap.jar*), který obsahuje pouze ty nejnutnější třídy nutné ke startu kontejneru. Ty samozřejmě nemůžou, mimo JavaSE, využívat žádné další API, jelikož by proměnná *classpath* musela odkazovat i na toto API. Tyto třídy slouží pouze k nastavení daných *Classloaderů* a následnému zavolání metody, která již kontejner opravdu spustí (tj. nastaví konektory, obsluhovače a vytvoří server). Důležité je, že tato metoda je volána až po nastavení oněch *Classloaderů*, a tak již smí být obsažena i v takové třídě, která již má plný přístup k celému API. *Classloadery* jsou totiž nastaveny tak, aby měly odkaz na všechny třídy, které kontejner potřebuje ke svému běhu (krom vlastního API to jsou také veškeré knihovny třetích stran, které API využívá). Další funkcionalitu, kterou mohou „*bootstrap.jar*“ třídy nabízet, je také ukončení kontejneru.

Doposud jsem také nezmínil, k jakému účelu jsou vůbec, servletovými kontejnery, specifické *Classloadery* vytvářeny? Důvod je jednoduchý. Tímto, kontejnery, mají zaručenu plnou kontrolu nad všemi třídami, které během svého chodu využijí. Kontejner tak může provádět například různé optimalizace, hotswap a v neposlední řadě také omezovat viditelnost jednotlivých tříd pro konkrétní části kontejneru, tj. například pro jednotlivé webové aplikace. Dále je vhodné říci, že proměnnou *classpath* a *bootstrap* třídy nastavují dávky, které jsou většinou obsažené ve složce *bin*. Dávky jsou typicky dvě, jedna pro start a jedna pro ukončení kontejneru. Také se často tvoří pro vícero operačních systémů.

Stejně jako v případě distribuce, ani zde nebude kontejner *Pike* výjimkou. Tedy také bude vytvářet vlastní *Classloader* a *bootstrap* třídy. Je však nutné přiznat, že mu vlastní *Classloader* zřejmě žádné výhody nepřinese, neboť nebude realizovat ani jednu ze zmíněných pokročilých vlastností. To je však v pořádku, jelikož je prozatím pouze *proof-of-concept*.

Subsystém pro start a ukončení kontejneru bude tedy obsahovat následující:

1. *bootstrap* třídy, které budou schopny nastartovat proces spouštění nebo také zahájit ukončení kontejneru
2. dávky, které budou používat koncoví uživatelé
3. API realizující skutečný start kontejneru

4.1.4.1 Bootstrap třídy V případě kontejneru *Pike*, bude *bootstrap.jar* obsahovat pouze jednu jedinou třídu, a to *pike.startup.boot.Bootstrap*. Jak jsem již zmínil, tato třída nebude

mít vůbec žádné vazby na ostatní API a bude tedy využívat pouze standardní JavaSE. To z toho důvodu, že při startu kontejneru je dávkami nahráván pouze onen *bootstrap.jar*.

Již víme, že posláním třídy *Bootstrap*, je pouze nastavit *Classloadery* a zavolat metodu pro opravdové nastartování kontejneru. Z tohoto důvodu je velmi jednoduchá, a tak jsem ji, téměř celou, umístil do přílohy C, konkrétně do podkapitoly C.2. Na něm můžete vidět, jak je kontejner startován. Třída také podporuje zastavení, a to prostřednictvím metody *stopServer(String)*. Díky její jednoduchosti, zde již nepřidávám žádný další popis. Pouze si, prosím, všimněte, odkud je nahráváno *Pike* API. Je nahráváno ze složky *lib*, která, jak již víme, slouží pro uchování knihoven v podobě JAR souborů. Kontejner *Pike* ji pro tento účel bude využívat také.

4.1.4.2 Dávky pro spuštění a ukončení kontejneru

Dávky jsou velmi jednoduché, neboť jen nastaví *classpath* proměnnou na zmíněný *bootstrap.jar*, předají *start*, respektive *stop* parametr a spustí JVM. Startovací dávka se bude nazývat *startserv* a je zobrazena na výpisu 14. Dávku pro zastavení jsem pojmenoval *stopserv* a její výpis lze vidět na výpisu 15. Obě dávky budou obsaženy ve složce *bin* společně s *bootstrap.jar*, který bude obsahovat zaváděcí třídu, která byla popsána v předešlé podkapitole.

```
%JAVA_HOME%\bin\java -cp .;bootstrap.jar pike.startup.boot.Bootstrap start
```

Výpis 14: Startovací dávka kontejneru *Pike*

```
%JAVA_HOME%\bin\java -cp .;bootstrap.jar pike.startup.boot.Bootstrap stop
localhost
```

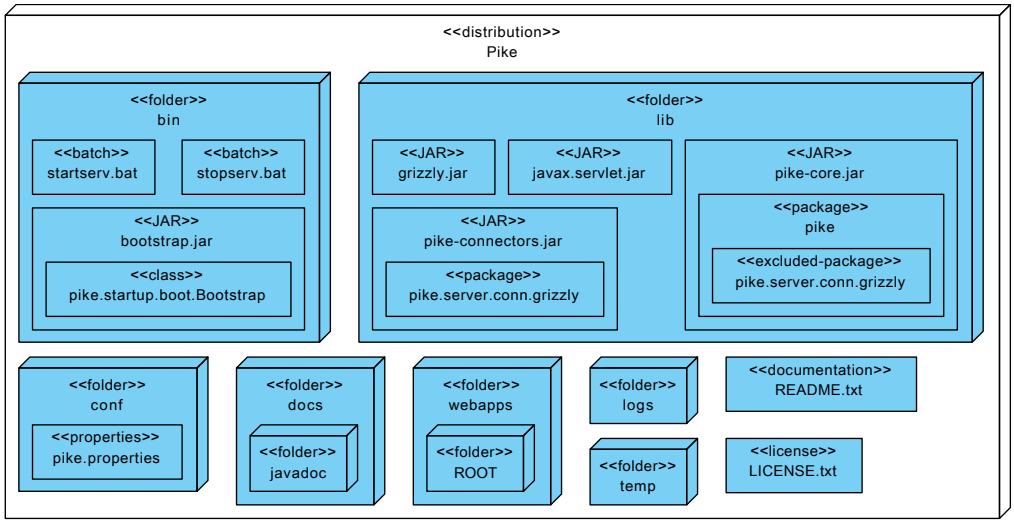
Výpis 15: Dávka pro ukončení kontejneru *Pike*

4.1.4.3 API realizující skutečný start kontejneru

Toto API bude taktéž velmi jednoduché, neboť bude obsahovat pouhé dvě třídy, a to třídu *pike.startup.Main*, respektive *pike.startup.ControlThread*.

Třída *Main* bude hlavní třídou realizující samotné nastartování, *ControlThread* bude pouze pomocné vlákno, které bude sloužit jen k naslouchání na příkaz k ukončení. Toto vlákno bude spouštěno třídou *Main*.

Již víme, že třída *Main*, respektive její metoda *start* je volána, výše zmíněnou, třídou *Bootstrap*. V této metodě je tedy provedeno vše, co je potřeba k úspěšnému chodu kontejneru, a pokud se podíváte na výpis C.3 v příloze C, uvidíte, co tato metoda skutečně obsahuje. Její kód je myslím dostatečně jednoduchý, proto ho zde nebudu popisovat. Stejně tak neuvádím kód třídy *ControlThread*, která pouze využívá JavaSE sockety a není ničím zajímavá.



Obrázek 8: Obsah distribuce kontejneru *Pike* po čtvrté iteraci

4.1.5 Aktuální obsah distribuce

Tato podkapitola nepopisuje žádné nové třídy a přidávám ji sem jen z toho důvodu, abych lépe vysvětlil aktuální obsah distribuce. Ta se nyní nachází ve své čtvrté iteraci, jež vznikla realizací subsystému pro start kontejneru. K popisu distribuce bylo využito služeb UML deployment diagramu, na kterém by mělo jít vše dobře vidět. Diagram je zobrazen na obrázku 8 a asi nepotřebuje žádný komentář. Snad jen uvedu, že složka s názvem *ROOT*, kerá je umístěna ve složce *webapps*, je servletová webová aplikace, která bude vždy dostupná pod kořenovým URL. Tato webová aplikace avšak nedefinuje žádné servlety, pouze obsahuje soubor *index.html*, který uživateli sděluje, že jeho *Pike* kontejner právě běží. Dále chci zmínit, že JAR soubor s názvem *javax.servlet.jar* obsahuje servletové API a stáhl jsem jej přímo z JCP stránek. Tento soubor je totiž distriubován spolu se servletovou specifikací.

4.2 Realizace servletové specifikace

Nyní již máme vše potřebné, abychom byli schopni realizovat samotnou servletovou specifikaci, respektive Servlet API. Toto stejně musíme, jelikož nám to nařizuje požadavek *REQ0006*.

Z prvků, které je možné definovat pomocí *deployment descriptoru*, budou podporovány pouze kontextové parametry, kontextoví posluchači, servlety a jejich inicializační parametry. Všechny ostatní části servletových webových aplikací podporovány prozatím nebudou. K tomuto kroku jsem se rozhodl jednak z *proof-of-concept* povahy kontejneru *Pike*, tak na základě toho, že servlety a kontextoví posluchači jsou, dle mého názoru, pro tvorbu mnoha webových aplikací naprosto dostačující. Také popis realizace servletového API bude jednodušší.

Stejně jako v předešlých podkapitolách, i zde, k popisu vlastní realizace, budu nadále využívat třídní a sekvenční diagramy. Jsem přesvědčen, že tento způsob je nejvhodnější.

4.2.1 Třídy pro integraci servletových webových aplikací do stávající architektury

Již víme, že architektura kontejneru *Pike* obsahuje rozhraní *IHandler*, které je serverem voláno pokaždé, když je, některým z konektorů, přijat nový požadavek. Je tedy zřejmé, že pro volání jednotlivých webových aplikací, je třeba vytvořit specializovaný obsluhovač, který, na základě URL přijatého požadavku, vždy zavolá odpovídající webovou aplikaci. Jak bude tento úkol v kontejneru *Pike* realizován, můžete vidět na obrázku 9. Obsluhovač se bude nazývat *ContextRequestDispatcher* a bude agregovat objekty implementující rozhraní *IContextHandler*. Toto rozhraní bude později rozšířeno třídou WebApp, která již bude realizací samotné webové aplikace, neboť *IContextHandler* definuje pouze cestu (část URL), na kterou je namapován. Důležité je, že toto rozhraní bude, stejně jako *ContextRequestDispatcher*, obsluhovačem. Tedy bude schopno obsluhovat klientské požadavky.

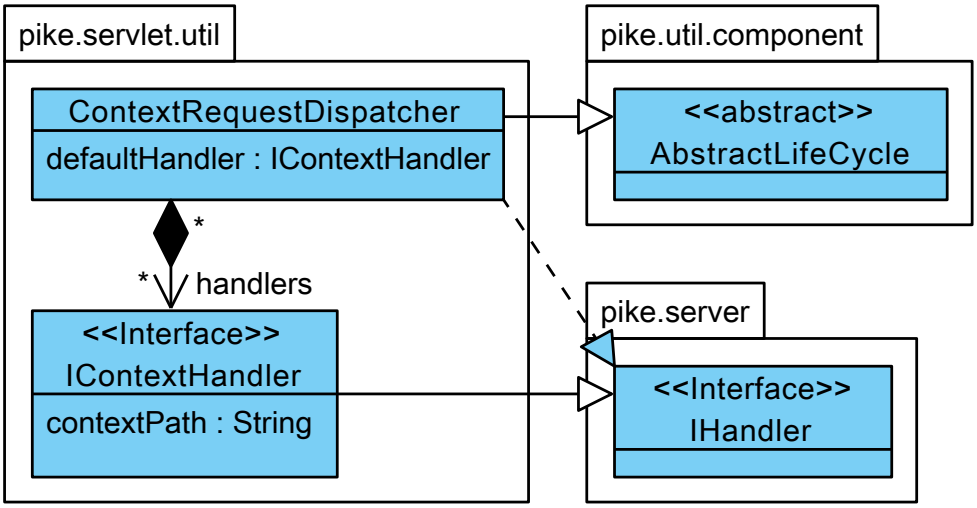
Chování těchto tříd za běhu kontejneru bude velmi jednoduché (proto zde také neuvádím žádný sekvenční diagram). Vždy, když server zavolá, na objektu *ContextRequestDispatcher*, metodu *handle*, bude, na základě URL přijatého požadavku, následně zvolen jeden z registrovaných *IContextHandler* objektů (popřípadě je vybrán výchozí *IContextHandler*, pokud žádný jiný danému URL neodpovídá), na kterém je poté zavolána metoda *handle*. Je nutné zmínit, že výběr objektů *IContextHandler* bude probíhat tak, jak je popsáno v servletové specifikaci v kapitole 12.1, tedy dle pravidla, že nejdelší prefix vyhrává.

Důvod, proč zde vytvářím něco jako *IContextHandler* a nespecifikuji zde přímo již webové aplikace, je ten, že chci zachovat jistou abstrakci, která je pro komponentní programování důležitá. V budoucnu tak může být *IContextHandler* rozšířen o další typy posluchačů a nemusí to být pouze servletová webová aplikace. Dále je nutné říci, že samotný *ContextRequestDispatcher*, objekty *IContextHandler* nijak nevytváří, ty jsou mu přiděleny. Toto je velmi důležité, neboť chci, aby toto API bylo od procesu nahrávání webových aplikací (deploymentu) naprosto odstíněno, jelikož pro něj není důležité, jakým způsobem jsou aplikace nahrávány. API, které bude deployment realizovat, bude, jak navrhnu později, samostatnou komponentou.

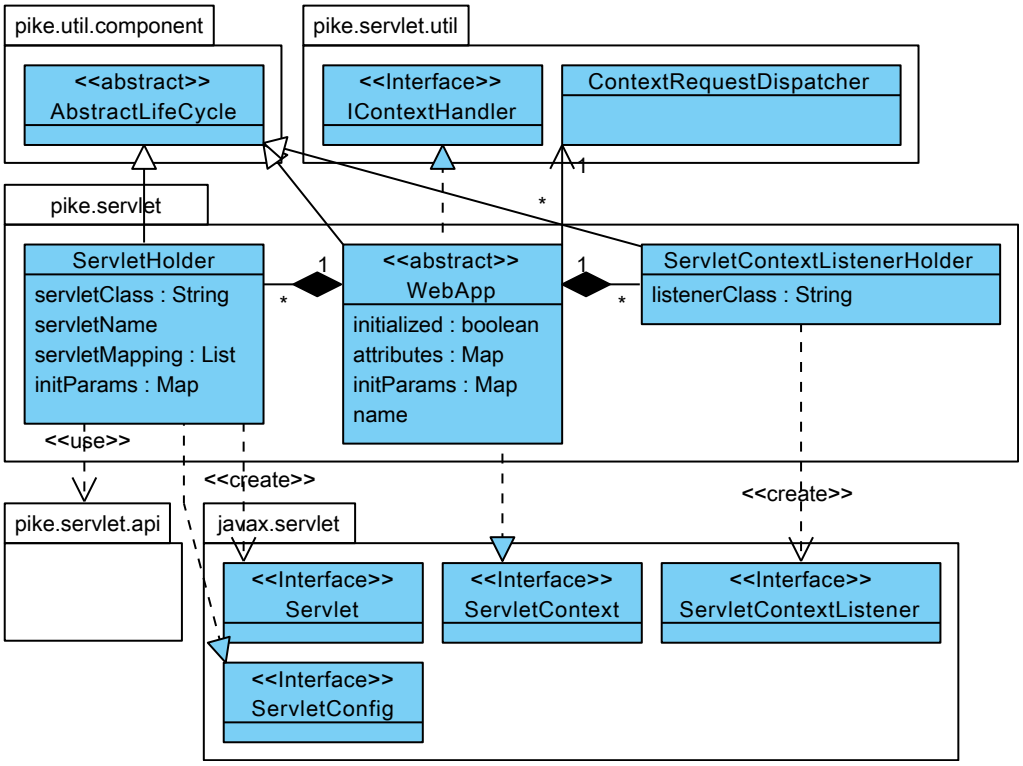
4.2.2 Realizace servletových webových aplikací

Každá servletová webová aplikace bude v kontejneru *Pike* realizována jedním objektem abstraktní třídy *WebApp*. Jak jsem již uvedl, tato třída bude implementovat rozhraní *IContextHandler*, tedy bude obsluhovačem, který se bude volat pokaždé, když URL požadavku (respektive jeho část) bude odpovídat kontextovému URL přidělenému této webové aplikaci. Jaké URL bude webovým aplikacím přidělováno, není pro toto API důležité, neboť se o tento úkol bude starat komponenta realizující deployment.

Na obrázku 10 je uveden třídní diagram, který zobrazuje ty nejdůležitější třídy, jež jsou nutné pro správný chod webových aplikací v kontejneru *Pike*. Jak můžete vidět, třída *WebApp*, krom rozhraní *IContextHandler*, také realizuje samotný *ServletContext* (z



Obrázek 9: Obsluhovač pro volání webových aplikací v kontejneru *Pike*



Obrázek 10: Třídy realizující servletovou webovou aplikaci v kontejneru *Pike*

něj ponechává pouze několik metod abstraktních, které musí realizovat až deployment. Jedná se o metody typu *getResource*.). Aby jej však mohla realizovat a také splňovala náležitosti servletové webové aplikace, musí mít odkazy na spoustu jiných tříd. Tyto třídy jsou zejména *ServletHolder* a *ServletContextListenerHolder*. Také musí obsahovat vícero atributů, jako například *contextPath*, *initialized*, *attributes*, *initParams*, *name* a další. Atribut *contextPath* již známe, *initialized* je booleovská hodnota vyjadřující, zda již byla aplikace inicializována (tj. zda již byli volání kontextoví posluchači), *attributes* je kolekce objektů nutná k realizaci rozhraní *ServletContext*, *initParams* je kolekce řetězců, která uchovává kontextové parametry a *name* je název webové aplikace.

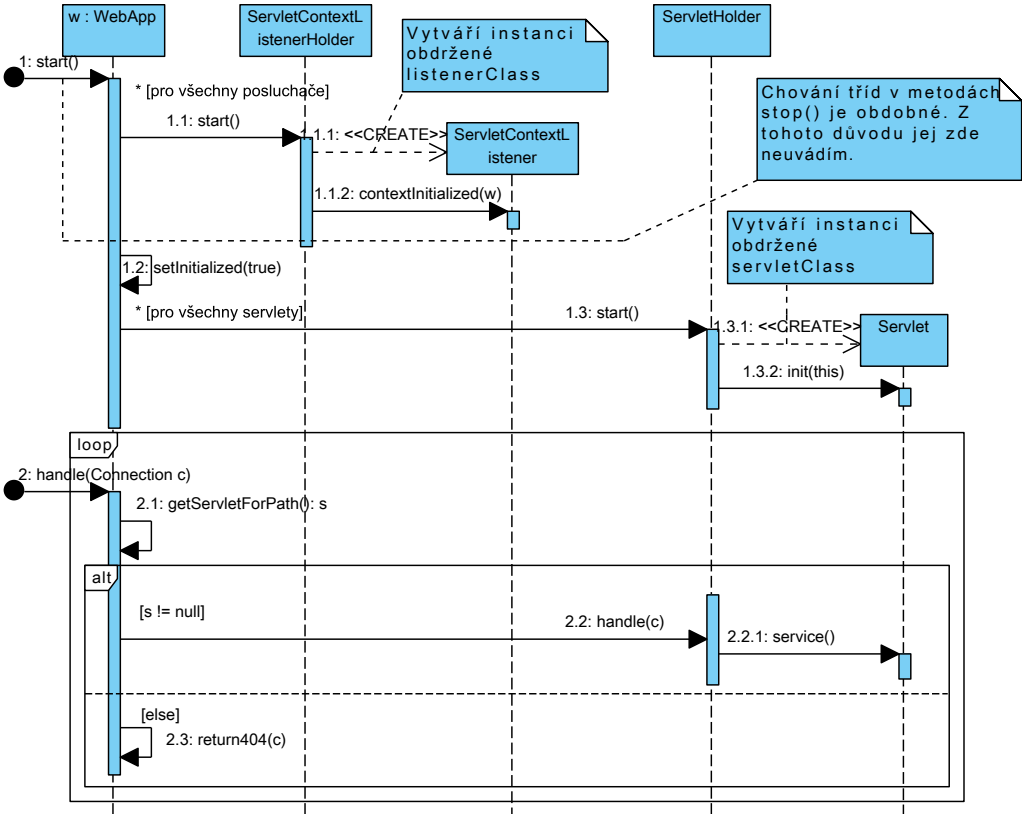
Jak jsem se již zmínil, třída *WebApp* dále agreguje objekty typu *ServletHolder*, respektive *ServletContextListenerHolder*. Tyto třídy nejsou ničím jiným, než reprezentací servletů, respektive kontextových posluchačů v kontejneru *Pike*. Třída *ServletHolder* je navíc také obsluhovačem, jelikož musí být schopna obsloužit klientské požadavky, a zároveň realizuje rozhraní *ServletConfig*. Tyto požadavky obsluhuje samozřejmě tak, že je předá servletu (tj. objektu typu *javax.servlet.Servlet*), který obaluje. Tato třída, ke své realizaci, využívá některé třídy z balíčku *pike.servlet.api*, který obsahuje implementaci dalších typů ze servletového API. Například realizaci rozhraní *HttpServletRequest*, či *HttpServletResponse*. Tento balíček zde však popisovat nebudu, jelikož není až tak důležitý a ani ničím zajímavý.

Jak třída *ServletContextListenerHolder*, tak třída *ServletHolder* má několik atributů. *ServletContextListenerHolder* má samozřejmě odkaz na svou *WebApp*, tak také řetězec, který specifikuje plně kvalifikovaný název třídy posluchače, který reprezentuje. Tento název je nastaven při nahrávání webové aplikace komponentou realizující deployment. Třída *ServletHolder* má také odkaz na svou webovou aplikaci, ale také například obsahuje kolekci inicializačních parametrů, název servletu, název třídy servletu a mapování servletu. Tyto atributy jsou také nastaveny při nahrávání webové aplikace.

4.2.2.1 Chování tříd realizujících servletovou webovou aplikaci

Výše zmíněné třídy jsou natolik signifikantní, že zde popíši také jejich chování za běhu kontejneru. Chování můžete přehledně vidět také na sekvenčním diagramu, který je zobrazen na obrázku 11.

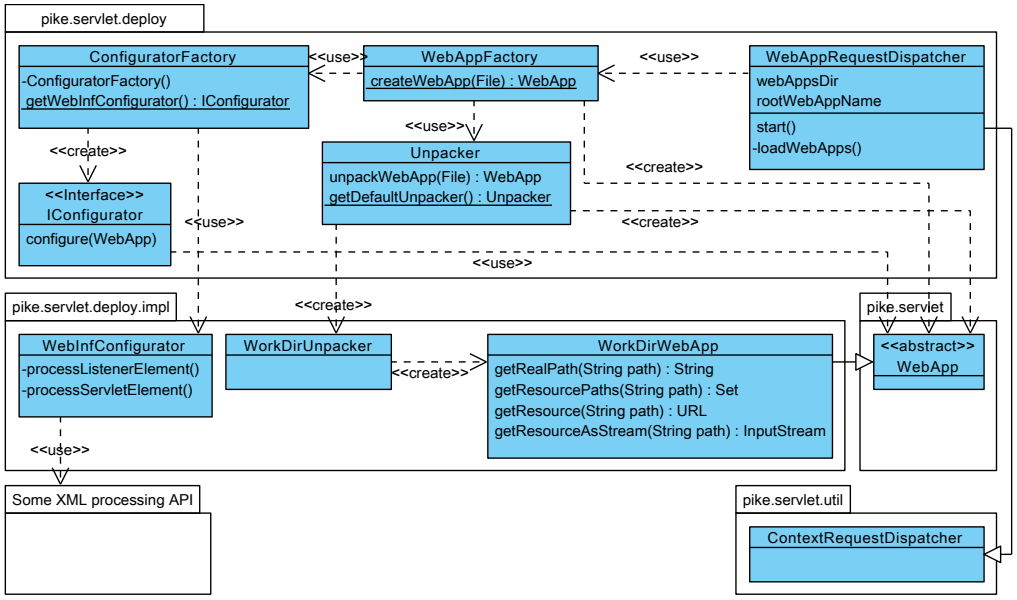
Tento diagram popisuje jak chování při startu kontejneru, tak během obsluhy každého požadavku. V průběhu startu kontejneru, je na každé webové aplikaci zavolána metoda *start* (ta je volána třídou *ContextRequestDispatcher*). V této metodě webová aplikace nejprve nastartuje všechny kontextové posluchače, taktéž voláním jejich metod *start*, poté změní svůj stav na inicializován a nakonec provede start všech servletů. Opět pomocí jejich metod *start*. Kontextoví posluchači, při svém startu, nejprve vytvoří instanci třídy typu *ServletContextListener*, jejíž jméno při nahrávání webové aplikace obdrželi, a následně na ní zavolají metodu *contextInitialized(ServletContextEvent)* s aktuální webovou aplikací. Obdobné kroky provedou také obalovači servletů. Tedy vytvoří instance tříd, které obdrželi, a následně na nich zavolají metodu *init(ServletConfig)*, které předají samu sebe, jelikož realizují také rozhraní *ServletConfig*. Těmito několika málo kroky je webová aplikace nastartována a je připravena obsluhovat požadavky klientů. Samozřejmě, pokud



Obrázek 11: Chování tříd realizujících servletovou webovou aplikaci v kontejneru *Pike*

během tohoto procesu nastane nějaká chyba, je buď zalogována, nebo je proces nahrávání webové aplikace zcela ukončen a aplikace nebude pro tento běh kontejneru funkční. Jak můžete dále vidět, diagram vůbec nepopisuje chování při ukončování kontejneru (tj. při volání metody *stop* na webových aplikacích). To z toto důvodu, že prováděné kroky při ukončování jsou velmi podobné krokům při startu. Namísto metod *start*, je na posluchačích a servletech volána pouze metoda *stop*, ve kterých jsou zase volány, namísto metod *contextInitialized(ServletContextEvent)*, respektive *init(ServletConfig)*, metody *contextDestroyed(ServletContextEvent)*, respektive *destroy()*.

Obsluha požadavků je taktéž velmi jednoduchá. Kdykoliv je, na webové aplikaci, volána metoda *handle(Connection)*, je nejprve získán servlet (respektive objekt třídy *ServletHolder*), který je namapován na URL, na nějž je obdržený požadavek směrován a poté je na něm zavolána metoda *handle(Connection)*, která je již zodpovědná za volání metody *service(ServletRequest, ServletResponse)*. Pokud však není žádný servlet nalezen, je namísto toho uživateli vrácena HTTP odpověď 404, která značí nedostupnost požadovaného zdroje. Proces výběru servletu na základě URL je popsán v servletové specifikaci v kapitole 12.



Obrázek 12: API pro nahrávání webových aplikací v kontejneru *Pike*

4.2.3 Nahrávání webových aplikací

Runtime reprezentaci webových aplikací již máme vytvořenou, nyní nám zbývá navrhnout API, které bude tuto reprezentaci vytvářet. Ta bude samozřejmě vytvářena při startu kontejneru, kdy je nutné, aby byly nahrány všechny webové aplikace, které aktuálně kontejner obsahuje.

Jak již víte, runtime API bylo navrženo tak, aby nebylo vůbec závislé na části, kterou je nutné realizovat nyní, tedy na deployment API. To z toho důvodu, abychom byli schopni webové aplikace vytvářet libovolnými způsoby. Nicméně v praxi jsou, servletovými kontejnery, aplikace vytvářeny většinou pouze jedním způsobem, a to načtením jejich definic ze složky z distribuce. Ani kontejner *Pike* nebude v tomto ohledu výjimkou a aplikace (ať již rozbalené nebo v podobě WAR souborů) bude načítat ze složky *webapps*, jež slouží k uchovávání všech webových aplikací, které má kontejner během svého běhu nabízet.

Jak bude deployment konkrétně realizován, lze vidět na obrázku 12. Spouštěcí třída deploymentu je obsluhovač *WebAppRequestDispatcher*, který mírně rozšiřuje, výše zmíněný, *ContextRequestDispatcher*. Jedinou změnou, kterou provádí, je, že přetěžuje metodu *start* (respektive *doStart*), ve které nejprve volá svou metodu *loadWebApps* a následně volá *doStart* předka, tedy *doStart* třídy *ContextRequestDispatcher*. Metoda *loadWebApps* je klíčová, neboť je to právě ona, která zahajuje veškerý deployment. Její zodpovědností je tedy vytvořit objekty typu *WebApp* pro každou podsložku složky *webapps*. Jakmile tyto objekty vytvoří, naplní jimi kolekci *IContextHandler* objektů, která je specifikována ve třídě *ContextRequestDispatcher*. Jelikož je následně volána také metoda *doStart* na této třídě, jsou všechny webové aplikace nastartovány, tj. je na nich zavolána metoda *start*, a jsou tak připraveny k použití.

WebApp objekty (tj. webové aplikace) jsou v metodě *loadWebApps* vytvářeny pomocí továrny s příznačným názvem *WebAppFactory*. Tato třída specifikuje jedinou metodu, která pro objekt typu *java.io.File* vytvoří implementaci třídy *WebApp*. Je tedy jasné, že realizace metody *loadWebApps* je velmi jednoduchá, neboť veškerou svou zodpovědnost přesouvá na tuto továrnu. Ovšem chování továrny se také příliš neliší (jak je to ostatně v OOP požadováno), neboť ta proces vytvoření kostry webové aplikace přenechává zase třídě *Unpacker* (v reálu její podtřídě *WorkDirUnpacker*), která již provádí opravdové vytvoření objektu typu *WebApp*, i když prozatím prázdného, tj. bez definovaných servletů, posluchačů atd.. Objekt typu *WebApp* je vytvářen pro *java.io.File*, který odkazuje na definici webové aplikace. Tento objekt může tedy odkazovat buď na složku, nebo na WAR soubor. Zodpovědností *Unpacker* objektů je vlastně vytvořit instanci abstraktní třídy *WebApp*. Třída *WebApp* je abstraktní z toho důvodu, že definuje celkem čtyři abstraktní metody, které definuje rozhraní *ServletContext*, a jež tato třída tedy neimplementuje. Všechny tyto metody se týkají načítaní zdrojů z dané webové aplikace. Je jasné, že webová aplikace může být uložena kdekoliv, a tak až třída, která ji opravdu načítá, je schopna vědět, jak tyto metody realizovat. Například třída *WorkDirUnpacker* webové aplikace neponechává ve složce *webapps*, ale kopíruje je (popřípadě extrahuje z WAR archívů) do složky *work*, která je taktéž v distribuci obsažena. To provádí z toho důvodu, že nechce jakýmkoliv způsobem měnit originální soubory, které byly nahrány samotnými uživateli. Složka *work* je pracovní složkou, do které si kontejner ukládá všechny soubory, se kterými potřebuje za běhu pracovat.

Poté, co je vytvořen prázdný objekt typu *WebApp*, je továrnou *WebAppFactory* tento objekt nakonfigurován. Konfigurace probíhá pomocí rozhraní *IConfigurator*, které obsahují jedinou metodu *configurate(WebApp)*, jež má za úkol nějakým způsobem nastavit obdrženou webovou aplikaci. Kontejner *Pike* prozatím obsahuje pouze jednu realizaci tohoto rozhraní, a to třídu *WebInfConfigurator*. Tato třída umožňuje nastavit předané aplikace dle jejich *deployment descriptorů*. Tedy je zodpovědná za vytváření a nastavování objektů typu *ServletHolder*, respektive *ServletContextListenerHolder*. K tomu, aby byla schopna *deployment descriptor* zpracovat, samozřejmě využívá API pro zpracovávání XML souborů.

4.3 Implementace hlavních novinek ze Servlet 3.0

V minulé podkapitole jsme dokončili pátou iteraci, která již obsahuje plně použitelný servletový kontejner, jež je schopen „rozběhnout“ servletové webové aplikace využívající servlety, kontextové posluchače a Servlet API do verze 2.5. Do distribuce kontejneru, k této iteraci, jsem tedy umístil webovou aplikaci „*HomePages*“, která je popisována v příloze A. Pokud se tedy podíváte na přiložené CD, můžete si ověřit, že kontejner *Pike* již takovouto aplikaci opravdu zvládne. Jedinou změnou, která je v této verzi „*HomePages*“ provedena, je náhrada autentizujícího filtru za autentizující servlet. To z toho důvodu, protože *Pike* doposud filtry nepodporuje.

Nyní je před námi tedy poslední úkol, a to realizace hlavních novinek z nejnovější verze servletové specifikace, které byly popisovány v kapitole 3. Jelikož je většina novinek menšího rázu, tedy nijak výrazně nemění architekturu kontejneru, ale pouze obohacují Servlet API, nebudu pro popis jejich realizace již nadále využívat „ptačí perspektivy“ a

většinu z nich vysvětlím pomocí výpisů zdrojových kódů. Pro realizaci každé novinky bude, stejně jako ve třetí kapitole, vyhrazena jedna samostatná podkapitola.

4.3.1 Realizace asynchronního zpracovávání požadavků

Přestože je asynchronní zpracovávání požadavků poměrně velkou změnou, svým implementačním rozsahem zase až tak obsáhlá není. Ovšem, co je na ní mírně složitější, je více-vláknové programování, na kterém, jak víme, je založena. To z toho důvodu, že objekt typu *AsyncContext* je pro všechny asynchronní cykly požadavku vytvořen jen jeden a může tedy k němu přistupovat několik vláken najednou. Naším úkolem je tedy zajistit jeho synchronizaci, respektive integritu jeho stavu.

Realizaci rozdělíme do několika fází:

1. Úprava tříd *WebInfConfigurator* a *ServletHolder* tak, aby bylo podporováno, zda daný servlet podporuje či nepodporuje asynchronní zpracovávání požadavků.
2. Realizace rozhraní *AsyncContext*.
3. Implementace metod třídy *pike.servlet.api.HttpServletRequest*, které se týkají asynchronního zpracovávání požadavků.

4.3.1.1 Podpora atributu asyncSupported K tomu, abychom byli schopni rozeznat, zda servlet podporuje či nepodporuje asynchronní zpracovávání požadavků, musíme nejprve přidat nový atribut do třídy *ServletHolder*, která v kontejneru *Pike* reprezentuje jednotlivé servlety.

To provedeme velmi jednoduše, prostě do této třídy přidáme novou instanční proměnnou, jejíž definici můžete vidět na výpisu 16. Proměnná je implicitně nastavena na hodnotu *false*, přesně tak, jak je požadováno v servletové specifikaci.

```
public class ServletHolder extends AbstractLifeCycle implements IHandler,
javax.servlet.ServletConfig {
    ...
    private boolean asyncSupported = false;
    ...
}
```

Výpis 16: Atribut *asyncSupported* třídy *ServletHolder*

Samotná proměnná nám však nestačí, musíme ji taktéž někde nastavovat. Její nastavení samozřejmě provedeme při vytváření objektů typu *ServletHolder*, tedy ve třídě *WebInfConfigurator*. Tato třída je zodpovědná za zpracovávání souborů *web.xml* jednotlivých webových aplikací a dle jejich obsahu následnému plnění těchto aplikací servlety a kontextovými posluchači.

Kód třídy je poměrně velký, nebudu ho zde tedy vypisovat. Pouze uvedu, že do této třídy bude přidána logika pro zpracování elementu *async-supported*, který se může

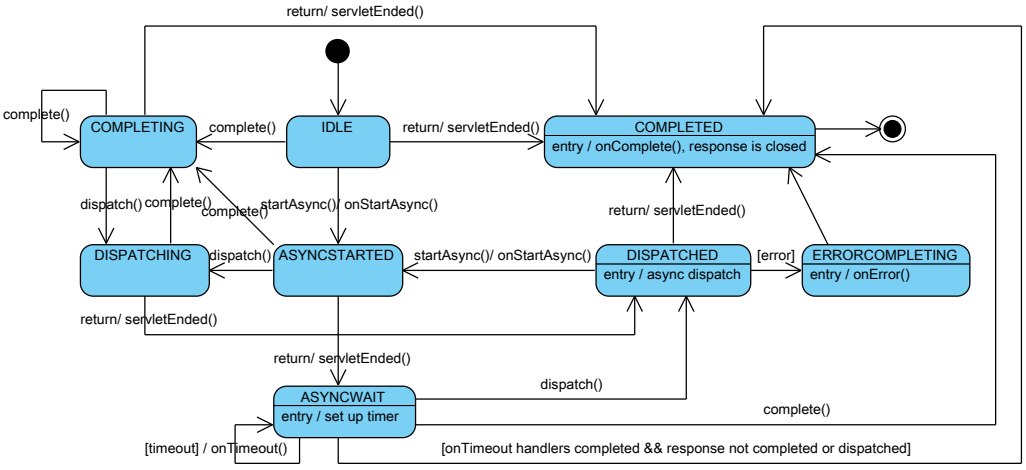
nacházet v elementu *servlet*. Obsahem tohoto atributu je booleovská hodnota, jež vyjadřuje podporu asynchronního zpracovávání požadavků.

4.3.1.2 Úprava třídy pike.servlet.api.HttpServletRequest Třída *pike.servlet.api.HttpServletRequest* reprezentuje v kontejneru *Pike* požadavky klientů, tedy je realizací rozhraní *javax.servlet.http.HttpServletRequest*. Pokud chceme, aby byla realizována podpora asynchronního zpracovávání požadavků, musíme v této třídě implementovat několik metod, které se asynchronního zpracovávání týkají. Jsou to metody *getAsyncContext()*, *isAsyncStarted()*, *isAsyncSupported()* a samozřejmě *startAsync()*, respektive *startAsync(ServletRequest, ServletResponse)* a na výpisu C.4 v příloze C je zobrazena jejich realizace.

Myslím, že implementace metod je dostatečně jednoduchá a přímočará. Pouze uvedu, že objekt typu *AsyncContext* není uchováván v požadavku, ale v objektu typu *Connection*, který je, jak již víme, vytvářen konektory pro každý nový požadavek. Je tedy pro požadavky unikátní, a proto jej lze využít pro uložení *AsyncContext* objektů. Dále je vhodné zmínit, že objekty *AsyncContext* jsou vytvářeny metodou *startAsync*, a to pouze v případě, že tato metoda nebyla doposud na tomto požadavku volána. Tedy, že je aktuálně zahajován teprve první asynchronní cyklus. A naposled, že *AsyncContext* objekty se po svém vytvoření nacházejí ve stavu *IDLE* a metodou *startAsync* přecházejí do stavu *ASYNCSTARTED*. Všechny stavy, ve kterých se může *AsyncContext* objekt nacházet, popíši v následující podkapitole.

4.3.1.3 Realizace rozhraní AsyncContext Realizaci rozhraní *javax.servlet.AsyncContext* bude obsahovat třída *pike.servlet.api.AsyncContext*. Je zbytečné zde uvádět její zdrojový kód, raději popíši její stavový diagram, který můžete vidět na obrázku 13. Ten je pro asynchronní kontext to nejpodstatnější, jelikož každý objekt typu *pike.servlet.api.AsyncContext* se v průběhu svého života může nacházet v několika stavech, které určují, co se s tímto objektem může v danou chvíli provádět. Je nutné zmínit, že stavový diagram popisuje pouze legální přechody, ostatní posloupnosti volání jednotlivých *AsyncContext* metod nejsou popsány, jelikož jsou zakázané a za běhu vyhazují výjimku. Úvodní vysvětlení stavů asynchronního kontextu je uvedeno v servletové specifikaci v kapitole 2.3.3.3.

Objekty typu *pike.servlet.api.AsyncContext* se po své inicializaci nacházejí ve stavu *IDLE*. Tento stav značí, že s tímto kontextem nebylo prozatím nijak manipulováno a je ekvivalentní situaci, kdy servlet vůbec asynchronní zpracovávání nevyužije. Jakmile je však zavolána metoda *startAsync*, kontext přejde do stavu *ASYNCSTARTED*. Tento stav znamená, že servlet využívá asynchronního zpracovávání a *Pike* v tomto případě, po ukončení metody *service* aktuálního servletu, neuzavírá odpověď. Kontext se tedy dostává do stavu *ASYNCWAIT*. Nicméně, pokud je ještě, před ukončením metody *service*, na kontextu volána metoda *complete*, popřípadě *dispatch*, kontext přechází do stavů *COMPLETING*, respektive *DISPATCHING*. Stav *COMPLETING* znamená, že jakmile metoda *service* skončí, bude odpověď uzavřena a všechen její obsah odeslán klientovi. Stav *DISPATCHING* zase znamená, že, po skončení metody *service*, se přejde do stavu *DISPATCHED* a zavolá se požadovaný servlet. Stav *DISPATCHED* tedy zahajuje nový asynchronní cyklus a je tedy téměř ekvivalentní stavu *IDLE*. Jediným rozdílem zde je,



Obrázek 13: Stavy třídy, která realizuje rozhraní *AsyncContext* v kontejneru *Pike*

možný přechod do stavu *ERRORCOMPLETING*. Do tohoto stavu se kontext dostane tehdy, pokud „dispatchovaný“ servlet vyhodí nezachycenou výjimku. V tomto případě se zavolají metody *onError* na všech posluchačích, kteří byli na tomto kontextu registrováni a následně se přejde do stavu *COMPLETED*. Stav *COMPLETED* je stavem konečným, který pouze volá metody *onComplete* na všech posluchačích a následně uzavírá odpověď. Poslední stav, který zbývá popsat, je stav *ASYNCWAIT*. Již víme, že do tohoto stavu kontext přejde tehdy, pokud servlet ve své metodě *service* zavolá metodu *startAsync* a poté již nezavolá metodu *complete*, či *dispatch*. Jakmile se do tohoto stavu přejde, je spuštěn časovač, který odpočítává nastavenou *timeout* dobu. Pokud se, po uplynutí této doby, kontext stále nachází ve stavu *ASYNCWAIT*, je zavolána metoda *onTimeout* na všech posluchačích. Pokud ani jeden z posluchačů v této metodě nezavolá metodu *complete* nebo *dispatch*, přejde se do stavu *COMPLETED* a odpověď je uzavřena automaticky. Tímto jsem tedy objasnil všechny stavy, ve kterých se může každý *AsyncContext* objekt nacházet. Jediné, co doposud zbývá popsat, je metoda *servletEnded()*. Tato metoda je definována taktéž ve třídě *pike.servlet.api.AsyncContext* a je volána, třídou ServletHolder, pokaždé, když skončí metoda *service*, ke kterému tento kontext patří. Tímto je kontext schopen, na tuto událost, reagovat a může provést vše, co potřebuje. Tedy zavolat metodu *complete*, provést *dispatch* nebo aktivovat časovač pro *timeout*.

4.3.2 Realizace API pro podporu anotací WebServlet a WebListener

Nyní je před námi celkově sedmá iterace, která má za úkol, do kontejneru *Pike*, přidat podporu pro anotace *WebServlet* a *WebListener*. Toto není těžký úkol, pouze stačí vytvořit další realizaci rozhraní *IConfigurator*, mírně modifikovat třídu *WebAppFactory*, kterou jsem popisoval již dříve, a upravit *WebInfConfigurator* tak, aby podporoval atribut *metadata-complete*, který je možné specifikovat v kořenovém elementu *web-app*. Pokud je tento

element nastaven na hodnotu *true*, je v dané webové aplikaci zakázáno hledat anotace pro definici servletů, filtrů a posluchačů.

Nový *IConfigurator* se bude nazývat *AnnotationsConfigurator* a jakým způsobem jej *WebAppFactory* do svého kódu integruje, je zobrazeno na výpisu 17. Popisovat modifikaci třídy *WebInfConfigurator* zde, stejně jako u realizace asynchronního zpracovávání požadavků, nebudu, jelikož třída *WebInfConfigurator* není ničím zajímavá, neboť jejím jediným úkolem je zpracovávat XML soubory.

```
public class WebAppFactory {
    ...

    public static WebApp createWebApp(File webApp) throws Exception {
        // rozbalime webovou aplikaci a vytvorime jeji prazdnou instanci
        WebApp webAppImpl = Unpacker.getDefaultUnpacker().unpackWebApp(webApp);
        // nastavime Classloader webowe aplikace, tedy URLClassloader, ktery
        // vidi na vsechny tridy z WEB-INF/classes a z JARu ve WEB-INF/lib
        setUpWebAppClassLoader(webAppImpl);
        // zpracujeme web.xml
        ConfiguratorFactory.getWebInfConfigurator().configure(webAppImpl);
        // zpracujeme vsechny anotace, ktere najdeme na tridach teto webowe aplikace
        // zde je tedy vyuzita trida AnnotationsConfigurator
        // anotace zpracovavame az po web.xml, jelikoz deployment descriptor
        // ma vyssi prioritu
        ConfiguratorFactory.getAnnotationsConfigurator().configure(webAppImpl);

        // vratime nakonfigurovanou webovou aplikaci
        return webAppImpl;
    }

    ...
}
```

Výpis 17: Integrace *AnnotationsConfigurator* do *WebAppFactory*

Implementace *AnnotationsConfigurator* je o něco zajímavější a je zobrazena na výpisu C.5 v příloze C. Jak můžete vidět, *AnnotationsConfigurator* vždy nejprve najde všechny třídy, které daná webová aplikace obsahuje (včetně těch tříd, které se nacházejí v JAR souborech ve složce WEB-INF/lib) a následně každou třídu skenuje pro *WebServlet* a *WebListener* anotace. Pokud je daná třída některou z těchto anotací anotována, je vytvořen nový objekt typu *ServletHolder*, respektive *ServletContextListenerHolder*, který je poté nastaven a následně přidán do webové aplikace. Zde je nutné přiznat, že pro skenování anotací je využito dočasného Classloaderu, který se vytvoří pomocí Classloaderu dané webové aplikace (tento Classloder je typu URLClassloader a má odkaz na všechny zdroje, které obsahují její kód), a metody *Class.forName(String, boolean, Classloader)*, která pro každou třídu vytvoří objekt typu *Class*, jež je nadále Classloaderem držen v paměti. Tento přístup je sice pohodlný (je možné využít API pro reflexi), ale není příliš efektivní, jelikož se zbytečně vytváří objekty typu *Class* pro každou třídu, kterou sebou aplikace nese. Sice je poté tento dočasný Classloader zničen, a může tedy proběhnout proces odebrání

všech vytvořených objektů typu *Class*, ale je zbytečně zabíráno mnoho paměti a času. Lepším řešením, které mě ale bohužel napadlo až po realizaci sedmé iterace, by bylo využít nějakou knihovnu pro zpracovávání bytecode. V tomto případě by se žádné *Class* objekty nevytvářely, pouze by se procházely samotné class soubory a hledaly by se v nich požadované anotace. Jelikož však *Pike* vyvíjíme iterativně, může se kód sedmé iterace refaktorovat v některé z dalších iterací.

4.3.3 Realizace sdílení zdrojů z JAR souborů

Implementovat sdílení zdrojů z JAR souborů dané webové aplikace (respektive z jejich *META-INF/resources* složek) bude velice snadné. Jak jsem již popisoval, kontejner *Pike* prozatím podporuje pouze jediný způsob nahrávání webových aplikací, a to ze složky *webapps*. K tomuto účelu využívá třídu *WorkDirUnpacker*, která rozšiřuje abstraktní třídu *Unpacker*. *WorkDirUnpacker* tedy provádí vytváření prázdných instancí typu *WebApp* pro objekty typu *File*, které odkazují buď na podsložku nebo WAR soubor ve složce *webapps*. Třída *WorkDirUnpacker* ve svém názvu obsahuje slova „work dir“ z toho důvodu, že každou webovou aplikaci kopíruje (popřípadě extrahuje) do pracovní složky v distribuci (konkrétně do složky *work*). Samozřejmě pro každou webovou aplikaci je vytvořena jedna samostatná podsložka.

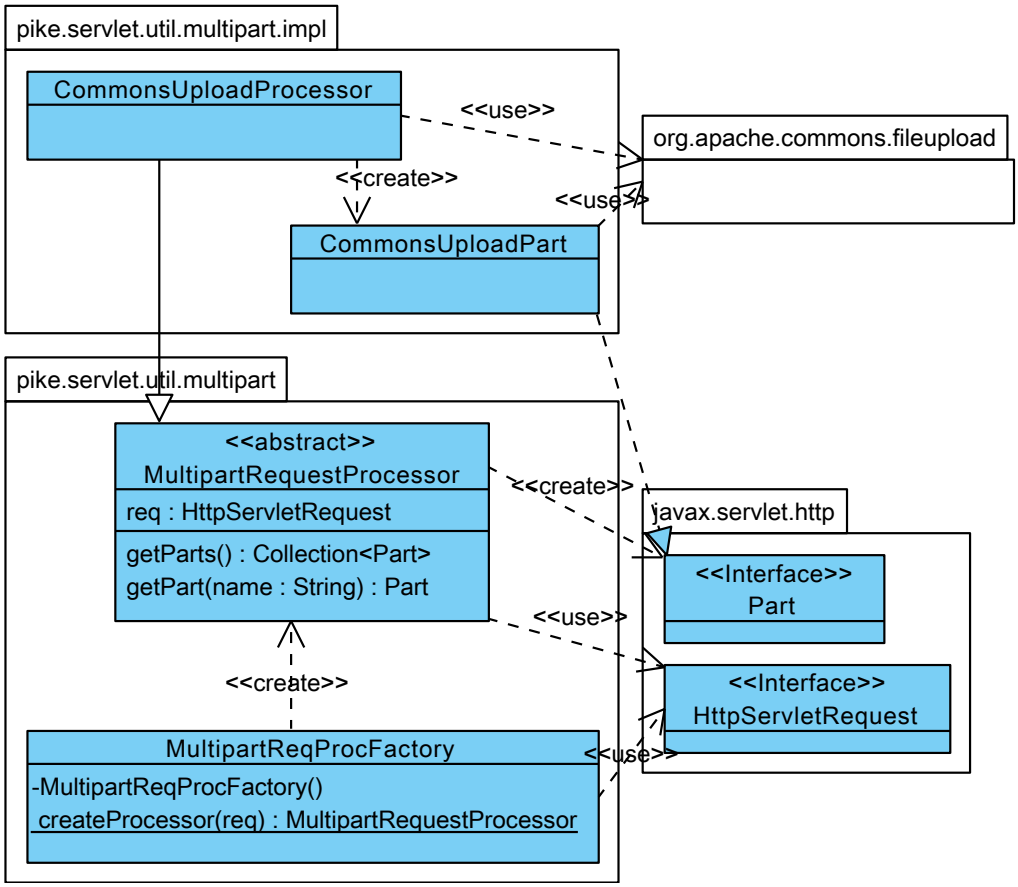
K tomu, abychom tedy realizovali sdílení zdrojů z JAR souborů, nám tuto třídu stačí mírně modifikovat. Upravit ji musíme tak, aby po nakopírování (či extrakci) dané webové aplikace, také prošla všechny její JAR soubory a nakopírovala obsah jejich *META-INF/resources* složek do hlavní složky dané webové aplikace, tedy do konkrétní podsložky složky *work*. Samozřejmě, při shodě názvů, je soubor ze složky *META-INF/resources* ignorován, neboť má nižší prioritu. Jakým způsobem bude tento úkol konkrétně realizován, je zobrazeno na výpisu C.6 v příloze C. Výpis obsahuje pouze fragmenty kódu, jelikož by bylo zbytečné, uvádět zde celý kód třídy *WorkDirUnpacker*. Ostatně, ten si lze prohlédnout ve zdrojových kódech osmé iterace. Stačí se podívat na třídu *pike.servlet.deploy.impl.WorkDirUnpacker*.

4.3.4 Implementace podpory požadavků typu multipart/form-data

V kapitole 3.12 jsem uvedl, že existuje celkem několik knihoven, které jsou určeny ke zpracovávání požadavků typu *multipart/form-data*. Protože je těchto knihoven více, API, které bude v kontejneru *Pike multipart/form-data* požadavky zpracovávat, navrhnu obecně. To z toho důvodu, aby bylo možné využívat libovolnou z těchto knihoven.

Tedy pro realizaci tzv. Parts API, které je u servletů zodpovědné za zpracovávání *multipart/ form-data* požadavků, musíme provést následující:

1. Navrhnout obecné API.
2. Pomocí obecného API implementovat *getPart* metody ve třídě *pike.servlet.api.Http ServletRequest*.
3. Realizovat obecné API pomocí jedné z knihoven.



Obrázek 14: API pro zpracovávání *multipart/form-data* požadavků v kontejneru *Pike*

Jak bude obecné API vypadat, je možné vidět na obrázku 14. Z tohoto obrázku lze dále vyčíst, že pro výchozí implementaci Parts API, bude využito služeb knihovny Commons FileUpload[41], která je jedním z projektů konsorcia The Apache Software Foundation. Hlavní třídou, která bude zodpovědná za zpracování požadavků, bude abstraktní třída s názvem *MultipartRequestProcessor*. Tato třída definuje dvě abstraktní metody, a to *getPart* a *getParts*, které nabízejí totožnou funkcionalitu jako stejně pojmenované metody v rozhraní *HttpServletRequest*. Je tedy zřejmé, že implementace těchto metod ve třídě *pike.servlet.api.HttpServletRequest* bude velmi snadná. Tato třída jen, s pomocí továrny *MultipartReqProcFactory*, získá objekt typu *MultipartRequestProcessor* (který za běhu bude typu *CommonsUploadProcessor*) a následně na něm zavolá jednu z metod *getPart*, respektive *getParts*. Jelikož je tato realizace velmi snadná, nebudu zde již uvádět její zdrojový kód. Stejně tak zde neuvedu ani implementace tříd *CommonsUploadProcessor*, respektive *CommonsUploadPart*, jelikož jsou většího rozsahu. Pokud přesto jejich kód chcete vidět, podívejte se, prosím, do zdrojových kódů deváté iterace.

[41] Tato knihovna je dostupná pod totou adresou: http://commons.apache.org/fileupload/

4.3.5 Realizace programové registrace servletů

Jak víme, programová registrace servletů je reprezentována metodami *addServlet*, *createServlet* a *getServletRegistration* a rozhraními *ServletRegistration*, respektive *ServletRegistration.Dynamic*.

Tedy pro její realizaci musíme nejprve vytvořit třídy, které budou tyto dvě rozhraní realizovat, následně implementovat, výše zmíněné, metody tak, aby tyto třídy využívaly a poté mírně upravit třídu *WebApp* takovým způsobem, aby byla schopna uchovávat objekty typu *ServletRegistration*. Doposud má totiž servlety reprezentovány pouze pomocí objektů typu *ServletHolder*.

Rozhraní *ServletRegistration*, respektive *ServletRegistration.Dynamic* bude v kontejneru *Pike* realizováno třídou *pike.servlet.api.ServletRegistration*, respektive *pike.servlet.api.DynamicServletRegistration*. Na realizaci těchto tříd není nic zajímavého, proto zde neuvedu ani jejich zdrojový kód (ten je, v případě zájmu, umístěn v desáté iteraci). Snad jen podotknu, že třída *DynamicServletRegistration* rozšiřuje třídu *pike.servlet.api.ServletRegistration* a obě dvě ke své implementaci potřebují pouze odkaz na objekty typu *ServletHolder*, respektive *WebApp*. Nic víc k jejich realizaci již potřeba není.

Jelikož se zbylé požadavky týkají už pouze třídy WebApp (ta totiž realizuje rozhraní *ServletContext*, ve kterém jsou definovány všechny metody pro programovou registraci servletů), její zdrojový kód je umístěn ve výpisu C.7 v příloze C, na kterém lze vidět, jak bude programová registrace konkrétně realizována. Kód ve výpisu je okomentován a myslím, že již žádné další vysvětlení nepotřebuje.

4.3.6 Realizace podpory objektů typu ServletContainerInitializer

Poslední novinkou z nejnovější verze servletů, kterou kontejner *Pike* realizuje (a vytvoří tak jedenáctou iteraci), bude podpora inicializátorů, tedy objektů implementujících rozhraní *ServletContainerInitializer*.

Jak víme, inicializátory se mohou nacházet ve dvou umístěních, a to buď na úrovni samotného servletového kontejneru, nebo přímo v rámci konkrétních webových aplikací. Dále specifikace nařizuje, že inicializátory, na úrovni kontejneru, se musí volat při startu každé webové aplikace a inicializátory, které se nacházejí v rozsahu jednotlivých webových aplikací, je povoleno volat jen při jejich startu.

Z těchto důvodů, bude kontejner *Pike* podporu inicializátorů realizovat takto:

1. Vytvoří třídu, která při své inicializaci načte všechny inicializátory na úrovni kontejneru.

2. Do této třídy přidá statickou metodu, která bude přijímat objekt typu *WebApp*. Jakmile se tato metoda zavolá, notifikují se všechny inicializátory na úrovni kontejneru, načtou se inicializátory předané webové aplikace a následně se zavolají i ty.

3. Upraví se metoda *start* třídy *WebApp* tak, aby vždy volala, výše zmíněnou, metodu. Tuto metodu bude samozřejmě volat před notifikací kontextových posluchačů.

Třída realizující podporu inicializátorů se bude nazývat *ServletContainerInitializerSupport* a nejdůležitější části jejího kódu můžete vidět na výpisu C.8 v příloze C. Na tomto výpisu není nic moc zajímavého, snad jen to, že se pracuje s classloadery kontejneru a jednotlivých webových aplikací a pro nalezení inicializátorů se využívá služeb třídy *java.util.ServiceLoader*.

5 Závěr

Hlavní cíle této práce byly dva. Prvním cílem bylo podrobně popsat ty nejpodstatnější novinky v nejnovější verzi servletové specifikace, tedy ve verzi s označením 3.0. Tím druhým pak byl návrh a následná realizace vlastního servletového kontejneru, který bude umožňovat využití některých novinek.

5.1 Přínos této práce

Oba hlavní cíle byly splněny. Novinky byly podrobně popsány v kapitole 3, přičemž byly vysvětleny nejenom ty nejpodstatnější z nich, ale byly uvedeny téměř všechny změny, které se v nejnovější verzi servletů udály. Opomenuto bylo pouze několik nepříliš podstatných vylepšení v servletové specifikaci a servletovém API. Každé novince byla věnována samostatná podkapitola, kde jako první byly objasněny ty nejhlavnější, které jistě budou zajímat velké procento programátorů servletových webových aplikací. Mezi takovéto změny nepochybně patří například asynchronní zpracovávání klientských požadavků, modularizace *deployment descriptoru* či definice servletů, filtrů a posluchačů pomocí anotací. Tyto tři změny byly popsány v kapitolách 3.1, 3.2, respektive 3.3. Domnívám se, že kapitola 3 podává přehledný a ucelený popis téměř všech novinek, v technologii Java Servlet 3.0, jež do těchto chvil není nikde jinde k dispozici.

Návrhu vlastního servletového kontejneru byla věnována kapitola 4. Ta byla dále rozdělena do tří podkapitol, kde v první byl nejprve podán návrh základní architektury, ve druhé, nad touto architekturou, proběhla realizace servletového API a nakonec ve třetí bylo implementováno několik novinek z nejnovější verze servletové specifikace. Tedy první podkapitola se zabývala výběrem webového serveru, distribucí kontejneru, konfiguračním subsystémem a realizací artefaktů a tříd, které dokážou kontejner spustit a ukončit. Druhá popsala návrh API, které v kontejneru realizovalo servletovou specifikaci do verze 2.5. Zabývala se tedy implementací hlavních prvků ze servletů a realizací subsystému pro nahrávání samotných webových aplikací. Konečně třetí podkapitola podala návrh a implementaci celkem šesti novinek z nejnovější verze servletů. Byly implementovány tyto novinky: asynchronní zpracovávání požadavků, anotace pro definici servletů, filtrů a posluchačů, sdílení zdrojů z JAR souborů, podpora požadavků typu *multipart/form-data*, programová registrace servletů a sdílení knihoven servletového kontejneru. I když je výsledný kontejner pouze takový *proof-of-concept*, již nyní dokáže bez větších problémů obsluhovat webové aplikace, které využívají pouze servlety, kontextové posluchače a API, výše zmíněných, šesti novinek. Přínosem čtvrté kapitoly je tedy realizovaný servletový kontejner, jehož každý zájemce může využít ke svým účelům, včetně implementace svého kontejneru.

5.2 Další vývoj

Poněvadž je vytvořený kontejner pouze *proof-of-concept*, možností pro jeho další vývoj je spousta. Samozřejmě prvním krokem je doimplementovat všechny náležitosti servletové specifikace a projít TCK pro JSR 315. Tzn. podporovat kompletní Servlet API a striktně

dodržet veškerá pravidla, která nám definuje servletová specifikace. Ve druhém kroku by bylo vhodné provést refaktorizaci stávajícího kódu a provést všemožné optimalizace. Po tomto kroku by již kontejner měl být velmi kvalitní, robustní a dostatečně rychlou realizací servletové specifikace. Třetím, a logickým, krokem by bylo kontejner postupně obohacovat o podporu dalších technologií a pomalu jej transformovat na aplikační server. Krom technologií, které je nucen každý JavaEE aplikační server podporovat[42], by kontejner mohl implementovat také například WebSockety či nabízet vylepšenou integraci pro projekty jako Terracotta[43] a Hibernate[44].

[42]Soupis těchto technologií pro JavaEE verze 6 je dostupný v rámci JSR 316.

[43]Terracotta je velmi zajímavou Java technologií pro tvorbu snadno škálovatelných aplikací. Její domovská stránka je dostupná pod tímto URL: http://www.terracotta.org.

[44]Hibernate je projekt, který nabízí objektový přístup k relačním datům. Více informací viz: http://www.hibernate.org.

6 Literatura

[1] HUNTER, Jason, CRAWFORD, William. *Java™ Servlet Programming*. Paula Ferguson. 1st edition. United States of America : O'Reilly & Associates, Inc., 1998. 528 s. ISBN 1-56592-391-X.

[2] BASHAM, Bryan, SIERRA, Kathy, BATES, Bert. *Head First Servlets and JSP™*. Brett D. McLaughlin; Louise Barr. 2nd edition. United States of America : O'Reilly Media, Inc., 2008. 912 s. Head First series. ISBN 978-0-596-51668-0.

[3] MORDANI, Rajiv, LUEHE, Jan, WILKINS, Greg. *Java™ Servlet 3.0 : Empowering Your Web Applications With Async, Extensibility and More* [online]. 3. 6. 2009 [cit. 2009-09-03]. Dostupný z WWW: <http://weblogs.java.net/blog/mode/servlet3.0/servlet3.0-javaone09.pdf>.

[4] XINYU, Liu. *Asynchronous processing support in Servlet 3.0 : Why asynchronous processing is the new foundation of Web 2.0* [online]. 19. 2. 2009 [cit. 2009-09-03]. Dostupný z WWW: <http://www.javaworld.com/javaworld/jw-02-2009/jw-02-servlet3.html>.

[5] FOWLER, Martin. *UML Distilled : A Brief Guide to the Standard Object Modeling Language*. 3rd Edition. United States of America : Addison-Wesley Professional, 2003. 208 s. ISBN 978-0321193681.

7 Přílohy

A Ukázková servletová webová aplikace

- Popis webové aplikace umístěn na CD v dokumentu s názvem *diplomova_prace_prilohy.pdf*.
- Implementace webové aplikace se nachází na CD v souboru *HomePages.war*.

B Změny v jednotlivých verzích servletové specifikace

- Příloha umístěna na CD v dokumentu s názvem *diplomova_prace_prilohy.pdf*.

C Zdrojové kódy hlavních tříd kontejneru Pike

- Příloha umístěna na CD v dokumentu s názvem *diplomova_prace_prilohy.pdf*.

D Webová aplikace využívající asynchronní zpracovávání požadavků (jedná se o asynchronní chat)

- Implementace webové aplikace se nachází na CD v souboru *async-request-war.war*.

E Servletový kontejner *Pike*

- Příloha umístěna na CD ve složce s názvem *KontejnerPike*. Složka dále obsahuje podložky pro každou iteraci a každá iterace jak spustitelnou distribuci, tak zdrojové kódy ve formě Eclipse projektu.